$\text{T}_{\!\text{E}}\!\text{X}_{\text{MACS}}$ as Authoring Tool for Formal Developments

Philippe Audebaud [1]

Lemme Project
INRIA
Sophia Antipolis, France

Laurence Rideau [2]

Lemme Project
INRIA
Sophia Antipolis, France

Abstract

We present an authoring tool for publication and dissemination of formal theories certified by a proof assistant. We have used TeXmacs, a "wysiwyg" editor for scientific documents which allows a connection with symbolic systems and we evaluate this tool on our formal developments certified using Coq.

Key words: Certification, Authoring Tool, Proof Assistant, Formal Development, Structured Document, LaTeX.

1 Introduction

We present a toolkit named TmCoq helpful for users who have validated some formal developments with a theorem prover assistant, such as Coq [?] and who want to ready their work for a scientific publication or online browsing. As such, we assume the user's main requirements are, firstly, to be able to present their work in the usual language for a mathematical article and, secondly, to preserve the property that the underlying development still remains certified by the prover. For the first goal, we offer a powerful tool which provides LaTeX quality typesetting for presenting the formal development and its documentation together. To preserve correctness of the development at each

[1] Email: Philippe.Audebaud@sophia.inria.fr
[2] Email: Laurence.Rideau@inria.fr

step, our approach to the second goal is to work always on the whole development, but to provide different *views* on this development. For documentation purposes, the user would prefer to be able to see the whole development, while the article is a selected view on the fully documented development, displaying only fragments of the whole development code.

A first solution that comes to mind consists in writing a LaTeX document. This is how one typically diffuses his results, inserting in places selected pieces of the formal development. This method has obvious drawbacks. The cut and paste process may introduce errors when the formal development had none. Any modification in the development requires the synchronization in the article, otherwise making the latter inaccurate or even wrong with respect to the certification step performed using the theorem prover. The inserted pieces of source code are introduced verbatim, which means that the syntax could be quite far from usual mathematical language and natural language as well. Notation and informal explanations are thus very different from the formal text. The theorem prover syntax may even change, introducing another source of misunderstanding.

To find a better solution to our problem, let us have a closer look at what makes up such a formal development in Coq. It consists of one or several files, called *scripts*, whose structure follows the usual one from a programming point of view. Declarations, statements, commands specific to the prover, etc. are part of the *vernacular sentences*. They form the *active part* of the script which means they are the only parts of the script that the theorem prover needs to know anything about. Any piece of documentation or explanation placed throughout the code can only be placed in the *comments* fields, which are irrelevant (i.e., *inactive*) from the prover's point of view.

Note that many documentation tools have been developed thus far for programming languages. Javadoc and Doxygen (among others) have made interesting contributions for this purpose. However, they are much too code-centric to be useful in our context. The users want to use LaTeX for writing, documenting, and publishing their work since a formal development consists of definitions, theorems, and proofs rather than pieces of programs. Nevertheless, a starting point is simply putting TeX commands within comments as part of the scripts.

On the Theorem Prover Assistant side, a LaTeX generation facility is offered by Pvs [?] for displaying formulas and proofs. It is based on an easy to use *substitution mechanism*. MetaPRL [?] provides (low level) formatting instructions based on the formatting library of its OCaml implementation language, together with Postscript and HTML output formats, but no LaTeX code generation as far as we know. For Isabelle, *isatool* [?] provides a batch LaTeX document generator. In Coq context, this functionality is performed by Coqdoc [?], which offers LaTeX and HTML as output formats from scripts. This tool makes a simple lexical analysis which is sufficient for the generation of hyper-links for identifiers and index tables for the various sorts of statements. The vernacular sentences are simply inserted *verbatim* in the generated document. Such a tool still fails to present the formal development fragments in a user-friendly syntax, as independent as possible from changes on the Coq side. Also, documentation and code still belong to different worlds. It is not possible to make forward references, insure uniform notation throughout the generated document. While the process has been automated, thus less prone to human errors, the result is still bound to the version of Coq used for the certification.

1.1 Objectives

The dissemination of formal developments (in scientific articles or by code distribution on the Web) is an important part of the development's life cycle, so we want to make a tool to help the user perform this task with the assurance that the correctness of the presented source code is preserved.

This tool must help the user to write code documentation (e.g., writing comments in programs) and it also must help the user to produce scientific articles by allowing code importation and by making it easy to add natural language explanations. We want to offer various output formats, for example, allowing the consulting and browsing of the commented code through a Web interface (i.e., the tool has to manage navigation links in the code) and offering all standard output formats used in scientific document publishing: LaTeX, HTML, Postscript, etc.

The tool must provide high-quality visualization, using notation that is familiar to the reader (e.g., using standard mathematical notation) as well as a user-friendly interface. It also must allow one to present the formal developments using a natural syntax and must avoid whenever possible the use of a given theorem prover's scripting language. That is, the interface should display $\forall a : \mathrm{nat}, \sqrt{a^2} = |a|$ rather than `(a:nat)(equal (sqrt  (power a (2))) (absolu a))`. The user must be able to define his own notation, and this notation must be kept consistent throughout the various parts of the document (i.e., keep the same notation in the explanations as in the imported code fragments).

Finally, to avoid a common source of errors in the translation from the source code to the presentation (typos are always unpleasant but are particularly annoying in the context of certified developments), the tool must allow the *automatic* importation of the source code (for example the theorems statements, the definitions, etc.) to preserve their correctness. This automatic translation should also make it possible to preserve links, connections, correspondences back and forth between the presentation and the scripts produced, and controlled by the prover.

Automatic translation preserves the visualized code's correctness as long as the user does not modify it. Here, one can import only fragments of the certified code in the final document. But, as soon as one needs to modify the imported code, to assure the correctness means that our tool has to manage the whole development (even if only fragments are displayed) either to allow an extraction of the development for certification outside of the tool or to allow an interactive certification inside the tool, communicating directly with the prover. Actually, unlike a fragment of programming code that can be considered independently of the rest of the code, a fragment of formal development can only be certified in the context of the whole development. Figure **??** shows the documentation process with the initial importation of the script, followed by the documentation steps, and finally the extraction into various formats in particular the Coq script extraction, allowing another cycle of certification.

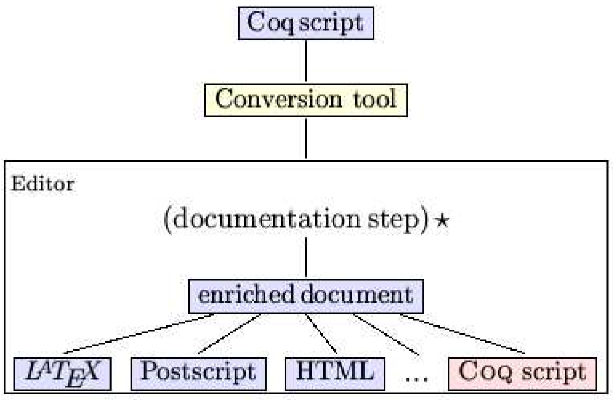


Fig. 1. Generation of the documentation from a Coq script.

The Coq script target is important. First, for compatibility reasons, we must provide a stand-alone version of the script, where the documentation fragments are inserted in the appropriate comment regions. Second, and more importantly, this target is evidence that one can extract from the enriched document the parts which concern the user formal development, this new script must be accepted by the prover afterwards.

1.2 Some examples

Our tool is not tightly coupled to a given theorem prover. In the future it should even be an independent tool. However, we begin with a case-study of our formal developments which are mainly done using the Coq theorem prover. Thus, the presented examples use Coq syntax.

Let us take a simple example. Note that in Coq concrete syntax `(X:A)B` stands for the the universal quantification $\forall X : A.B$, `~A` for the logical negation A, while `(EXT x:C | P)` is meant for the existential connective $\exists x : C.P(x)$.

```
Welcome to Coq 7.4 (Feb 2003)
Coq < Classical_Pred_Type.
Coq < Check not_all_ex_not.
not_all_ex_not :
  (U:Type; P:(U->Prop))~((n:U)(P n))->(EXT n:U|~(P n))
```

Therefore, instead of displaying the type of `not_all_ex_not` as

```
(U:Type; P:(U->Prop))~((n:U)(P n))->(EXT n:U|~(P n))    (1)
```

one would rather see

$$\forall \mathsf{U} : \mathsf{Type}; \mathsf{P} : \mathsf{U} \rightarrow \mathsf{Prop}.\ \neg(\forall n : U.(Pn)) \rightarrow (\exists\ \mathrm{n} : \mathrm{U}.\ \neg(Pn)) \tag{2}$$

Eventually, one would even hope to have the possibility of pretty printing this statement in natural language. For instance, instead of the raw Coq lemma:

```
Lemma Set_nrootIR :
  (n:nat)(lt (O) n)->(c:IR)(Zero [<=] c)
              -> {x:IR & (Zero [<=] x) * (x[^]n [=] c)}
```

which depends heavily on the `Notation` enhancement mechanism, one prefers to see

Lemma 1.1 (Set_nrootIR) $\forall n \in \mathbb{N}.0 < n \rightarrow \forall c \in \mathbb{R}.0 \leq c \rightarrow \exists x \in \mathbb{R}.0 \leq x \wedge x^n = c$

or, even better,

Lemma 1.2 *For every positive integer n and non-negative real number c, there exists a non-negative real number x such that $x^n = c$.*

This last presentation is easier to read, hence it provides a better presentation for the formal statement within an article aimed for diffusion.

This raises several questions about pretty printing existing documents. Is the verbatim output, as shown in (**??**), sufficient to make it possible to display (**??**) or the two lemmas shown above? It should be clear that nothing is possible unless the text from (**??**) is *re-parsed* in order to grab enough information on the actual structure of this code fragment. The next question is: what is the best internal representation (abstract syntax) which is capable of providing the information required to make the various displays possible? And at what cost? Finally, what is the best way to generate the various styles and the formats?

The user expects other features as well. For instance, he might want to modify imported fragments to make them more readable or easier to understand. Thus, variable renaming, hypotheses reorganization, or other changes at the presentation level should be possible while keeping the formal development correct with respect to the underlying proof assistant. Such demands are reasonable since one cannot expect any automatic generation from a formal development to provide any article-ready text. As soon as interactive editing introduces changes with respect to the initial development, there is a possibility for introducing errors. It is these errors we would like to avoid with our tool.

2 Software Design Decisions

To build our tool, we have decided to use TeXmacs to write scientific articles and we have developed an extension for handling Coq input and output. The resulting system named TmCoq, implements the functionalities described in figure ??, where *editor* is TeXmacs. Exportations to and translations between the different formats (TeXmacs, PostScript, ...) are handled by the native TeXmacs system, but TmCoq needs to have control on the macros expension mechanism to avoid the lost of the structure information. This is the prerequisite for LaTeX exportation as well as for re-exportation toward Coq script syntax from a TeXmacs document (see section ?? for more details).

Regarding the Proof Assistant issue, we restrict ourselves to Coq developments for experimental purposes, with the long-term objective to build a tool adaptable to various theorem prover systems.

2.1 The scientific editor GNU TeXmacs

Quoting its documentation page [?], *"GNU TeXmacs is a free scientific text editor, which was both inspired by TeX and GNU Emacs. The editor allows you to write structured documents via a wysiwyg (what-you-see-is-what-you-get) and user friendly interface. New styles may be created by the user. The program implements high-quality typesetting algorithms and TeX fonts, which help you to produce professionally looking documents."*

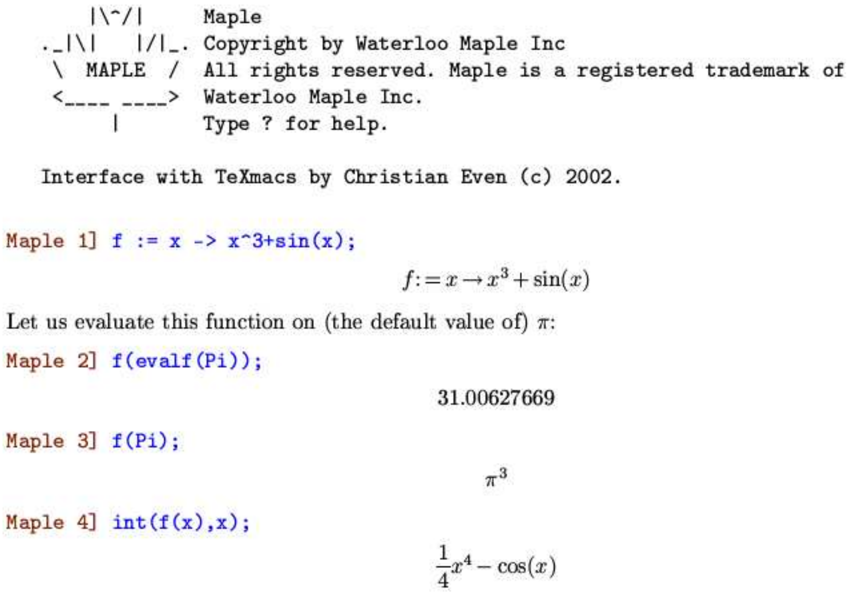
We definitively want to offer the user a way to enrich their formal development with convenient documentation or informal explanations. Typically after the automated importation step of a Coq script the user will do some interactive editing to prepare for dissemination of results. LaTeX exportation is satisfactory as a scientific document format. However, we must be able to recover the script to certify *a posteriori* the embedded code. Using LaTeX as the generic format for the script leads to a substantial loss of structure information. The resulting document is (enriched) text, while on the Coq side, a script is a program, where documentation is in comments. Therefore, the crucial distinction between the *active* parts of the script (the vernacular commands mainly) and the *inactive* parts (the interleaved comments-as-documentation fragments) is lost in the conversion process if we use LaTeX as the generic format. Although this could be handled to some extent with the help of tricky annotations within the LaTeX end-of-line based comments, this is not satisfactory. The TeXmacs editor deals with structured documents, which is precisely what we want.

As far as presentation style is concerned, the user expects to be allowed to customize the look, on file basis, on whole development basis, or even depending on whether the resulting document is about to be browsed or printed on paper. TeXmacs fulfills this criterion without requiring any special or additional code.

Moreover, TeXmacs also supports the Guile/Scheme extension language, so that it is easy to customize the interface and write extensions to the editor at the user level. Browsing is possible between distinct files as well as through the web. Automatic content generation includes indices, tables, figures, and bibliographies. TeXmacs documents are saved as structured text files using a simple notation system which helps automatic (machine-driven) search. Therefore, as a whole, TeXmacs presents many attractive features which make it an interesting candidate in the context of authoring scientific documents.

2.1.1 Communication with computer algebra systems

Among our motivations, interactivity with the theorem prover assistant was also required. As such, TeXmacs provides a *session mode* that also offers the same combination of editing and command execution as Emacs does for many computer algebra systems, the *operating system shell*, and a *Guile/Scheme top-level*. Figure **??** presents a short session with Maple.



```
    |\^/|     Maple
._|\|   |/|_. Copyright by Waterloo Maple Inc
 \  MAPLE  /  All rights reserved. Maple is a registered trademark of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.

Interface with TeXmacs by Christian Even (c) 2002.

Maple 1] f := x -> x^3+sin(x);
```

$$f := x \to x^3 + \sin(x)$$

Let us evaluate this function on (the default value of) π:

```
Maple 2] f(evalf(Pi));
```

$$31.00627669$$

```
Maple 3] f(Pi);
```

$$\pi^3$$

```
Maple 4] int(f(x),x);
```

$$\frac{1}{4}x^4 - \cos(x)$$

Fig. 2. A sample session with Maple

We see that the presentation facilities are automatically available for the output generated by Maple: the TeXmacs interface interprets the LaTeXoutput generated by Maple. Note that the session is allowed at any place within the document. Actually TeXmacs permits editing commands to be applied during the session, multiple interruptions, as well as the interleaving of computations and explanations.

This session mode benefits to some extent from the standardization of mathematical notation and the relative simplicity of the concrete syntax found in Computer Algebra Systems (CAS). Also, it must be emphasized that LaTeX output is offered by Maple [?], Mathematica [?], and other CAS, which shows that the CAS must be adapted in order to gain richer output. Finally, note that the session mode is closer to the interaction the user gets with a calculator or a Unix shell than what is expected for developing proofs with the help of a Theorem Prover Assistant (TPA).

2.2 On the COQ *side*

To go further than COQDOC, our work proposes a solution to offer richer output formats to COQ. So we need to do more work than a simple lexical analysis. At least a syntax analysis is required. We will show that this is actually sufficient. However before this, we have to choose between writing another parser for our purposes, or benefiting from the existing one present in the COQ source. There are two major arguments to take into consideration. First, COQ is a research tool, still subject to changes, even with respect to its concrete syntax. Second, the syntax allowed at the formula level requires a very complex parser. Therefore, the necessary synchronization would already suggest that we develop our tool as part of the COQ source tree rather than another stand-alone program. But we can do better than just sharing the parser. Our code can also be used to interface COQ with any external tool that requires exchanging data with COQ, for example PCOQ [**?**].

2.2.1 Formulas

Consider the following example, whichever notation one might choose, the internal structure for the above term `not_all_ex_not` is a tree:

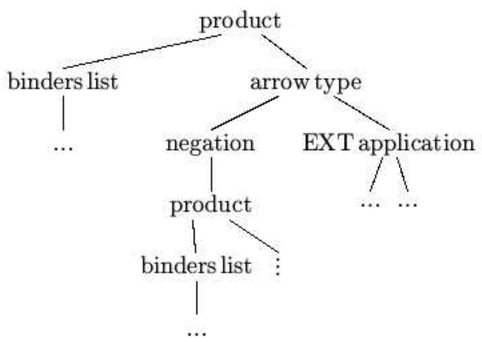


Fig. 3. Tree representation for `not_all_ex_not`

and the corresponding representation in TEXMACS format is

```
∀{binder_list|P : U → Prop}. {arrowc|{N~_|~|(∀n : U. {appc|P n})}|({NEXT_:_|_|
EXT |n|:||U|||¬(Pn)})}
```

The binder_list, arrowc, and appc nodes should be obvious from the above tree. The `N~_` and `NEXT_:_|_` come from the extendible `Notation` command introduced in the last version of COQ that permits the kind of enhancement that formula (**??**) uses for the verbatim output. While this is not at all necessary in our tool (see RSA example below), at this stage of our development we decide to represent these nodes as special nodes in the generated structure.

2.2.2 Inductive definitions

Coq provides specific outputs in many contexts, not only for formulas. Let us have a look at the `Inductive` vernacular command. The Coq standard library provides many examples. From `Datatypes` unit, the sum of two sets $A + B$ is inductively defined as

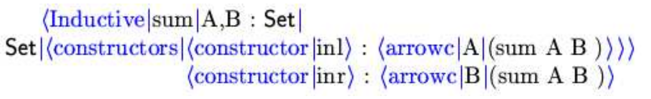
```
Inductive sum [A,B:Set] : Set
    := inl : A -> (sum A B)
     | inr : B -> (sum A B).
```

It is worth noting that the concrete syntax is usually chosen on the basis of

(i) its ability to be read easily by the human reader,

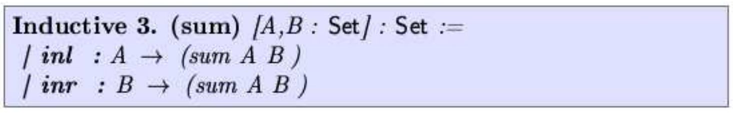
(ii) and its ability to be read automatically, hence understood by the computer.

Come to think of it, both points would hold if we switched humans and computers as well! The important thing to keep in mind is that for both points it should be possible to infer from the concrete representation that an inductive definition is *structurally* made of a finite sequence of *named* items (`inl`, `inr` in this example) as constructors with respective type information. Also, the syntax [A,B: Set] means that sets `A` and `B` are *parameters* in the definition, the result being a set as the fragment ": Set" shows. Therefore, an inductive definition comes with its name, its parameters, the resulting type, and a finite list of constructors.



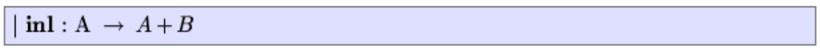
```
⟨Inductive|sum|A,B : Set|
Set|⟨constructors|⟨constructor|inl⟩ : ⟨arrowc|A|(sum A B )⟩⟩⟩
                  ⟨constructor|inr⟩ : ⟨arrowc|B|(sum A B )⟩
```

So it suffices to grab these fields as part of the structure to be able to display the inductive definition in various styles. A straightforward representation in $\text{T}_\text{E}\text{X}_{\text{MACS}}$ would be:



Inductive 3. (sum) *[A,B :* Set*] :* Set *:=*
*| **inl** : A → (sum A B)*
*| **inr** : B → (sum A B)*

A simple variation, closer to mathematical notation could be:



| **inl** : A → $A+B$

The user is free to decide globally or on a local basis among all these representations. Even more, switching between them is also available within the $\text{T}_\text{E}\text{X}_{\text{MACS}}$ interface.
The same result can be obtained in LaTeX from the corresponding source:

```
\Inductive{sum}{A,B : {\sort{Set}}}{{\sort{Set}}}{
  \begin{constructors}
    \constructor[inl] : ${\arrowc{A}{{\appc{sum A B}}}}$
    \contructor[inr]  : ${\arrowc{B}{{\appc{sum A B}}}}$
```

```
   \end{constructors}}
```

with the help of the appropriate macro definitions.

The more structure Coq is required to produce, the more freedom the user gets on their side to get the appropriate notation. As an example, one might want to present such inductive definition with the help of inference rules. In the Coq user manual (section 4.5.1), one finds

```
(* Lists of elements of set A *)
Inductive list [A : Set] : Set :=
   nil : (list A) | cons : A -> (list A) -> (list A).

(* Length for such lists *)
Inductive Length [A:Set] : (list A) -> nat -> Prop :=
   Lnil : (Length A (nil A) O)
  | Lcons : (a:A)(l:(list A))(n:nat)
           (Length A l n)->(Length A (cons A a l) (S n)).
```

Depending on the context, presentations like

$$(\text{Lcons})\frac{a : A, l : \text{list}(A), n : \mathbb{N} \vdash (\text{Length}\ \ A\ l\ n)}{a : A, l : \text{list}(A), n : \mathbb{N} \vdash (\text{Length}\ \ A\ (a.l)\ (n+1))}$$

or

$$(\text{Lcons})\frac{\text{length}_A(l) = n}{\text{length}_A(a.l) = n+1}$$

when the context is clear enough, could be desired *without any change* in the underlying document. These representations raise no technical difficulty. Rather, they require more decomposition in the Coq structure representation output. In LaTeX, the constructor types for the case of inductive definitions could be prepared along the lines of this example:

```
\constructor[Lcons]
   \inferrule[a:A, l:(list A), n:nat]
                   {(Length A l n)}{(Length A (cons A a l) (S n)}
```

The section **??** addresses other places where such output questions have to be dealt with on the Coq side.

3 Utilization

The TmCoq system consists in several components:

- Coq which gives us parsing of scripts for free. Actually, our version is modified to allow for generation of TeX$_{\text{MACS}}$ format.
- A set of TeX$_{\text{MACS}}$ styles and macros for the rendering. The user is free to customize this package, the same way as LLNCS class does with respect to the standard LaTeX article class.

– A set of Scheme scripts to accomodate the LaTeX and Coq scripts exportation filters.

The last two components are available from the TeXmacs user interface. Reusing LaTeX macros in the TeXmacs is possible through the import facility. Conversely, exportation of TeXmacs macros to LaTeX macros or environments is not yet offered by TeXmacs.

For documentation purpose, our modified Coq distribution provides a new wrapper for the `coqtop` program, named `tmdoc`. A generic utilization consists in running the command: `tmdoc MyScript.v` which results in the TeXmacs file `MyScript.tm`. Other output formats directly available from `tmdoc` are LaTeX, Scheme. However, as shown in figure (**??**), these formats together with HTML and Postscript are also available by editing `MyScript.tm`.

Let us illustrate further through two complete examples in the two following sections.

3.1 The Coq *Standard Library*

Let us have of look at the very beginning of the `theories/Init/Wf.v` file from the Coq standard library:

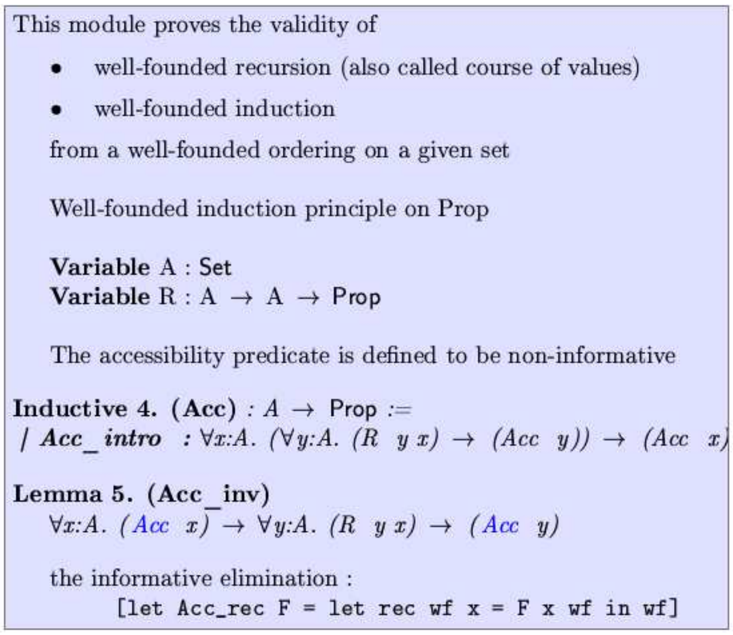
```
(*i $Id: Wf.tm,v 1.1 2003/05/27 11:30:57 paudebau Exp $ i*)

(** This module proves the validity of
     - well-founded recursion (also called course of values)
     - well-founded induction
    from a well-founded ordering on a given set *)
Require Logic.
Require LogicSyntax.

(** Well-founded induction principle on Prop *)

Variable A : Set.
Variable R : A -> A -> Prop.

 (** The accessibility predicate is defined to be
     non-informative *)

 Inductive  Acc : A -> Prop
    := Acc_intro : (x:A)((y:A)(R y x)->(Acc y))->(Acc x).

 Lemma Acc_inv : (x:A)(Acc x) -> (y:A)(R y x) -> (Acc y).
  NewDestruct 1; Trivial.
 Defined.
 (** the informative elimination :
     [let Acc_rec F = let rec wf x = F x wf in wf] *)
```

Coq comments have the form `(*...*)`. The documentation fragments are identified by the special comment construction `(**  ...*)` . This construction follows Coqdoc *almost text* conventions, which helps in writing structured pieces of text without knowing too much of TEX or HTML syntax and more importantly in a independent syntax way. Therefore, to keep compatibility with existing documentation fragments in Coqdoc style, we have added to `tmdoc` another output format to make these tags understandable by $\TeX_{\text{MACS}}$. This comment is thus translated into

```
(** texmacs: This module proves the validity of
<\itemize>
<item> well-founded recursion (also called course of values)
<item> well-founded induction
</itemize>
   from a well-founded ordering on a given set  *)
```

One can see that we have extend the class of comments-as-documentation by the syntax `(** filter: ... *)` where `filter` ranges over `coqdoc`, `scheme`, `texmacs`, `latex` and defaults to `coqdoc` for compatibility's sake.

As for translation of the vernacular commands, our tool goes far beyond Coqdoc capabilities, but at the cost of requiring `coqtop` to perform both parsing and $\TeX_{\text{MACS}}$-aware structured output, as explained before.



This module proves the validity of

- well-founded recursion (also called course of values)
- well-founded induction

from a well-founded ordering on a given set

Well-founded induction principle on Prop

Variable A : Set
Variable R : A → A → Prop

The accessibility predicate is defined to be non-informative

Inductive 4. (Acc) *: A →* Prop *:=*
*| **Acc_intro** : ∀x:A. (∀y:A. (R y x) → (Acc y)) → (Acc x*

Lemma 5. (Acc_inv)
∀x:A. (Acc x) → ∀y:A. (R y x) → (Acc y)

the informative elimination :
`[let Acc_rec F = let rec wf x = F x wf in wf]`

Note that `Require` commands and proofs do not appear, although they have been translated and still remain present in the generated document. We provide flags that allow one to modify the material presented, which comes in handy for full script rendering or to show the mathematical contents of this file. Needless to say, $\TeX_{\text{MACS}}$ offers many more options, either through the existing environment, macros, or GUILE scripting.

Following these brief explanations, the full COQ Standard Library is generated automatically by our documentation tool as a bunch of $\TeX_{\text{MACS}}$ files which can be browsed within the editor. The blue items `Acc` (in lemma 5) represent *hyperlinks* to the definition declaration `Inductive 4.`. Our contribution consists in a complete set of $\TeX_{\text{MACS}}$ files accessible online from within the editor and browsable in the same way as the web documentation on the COQ site.

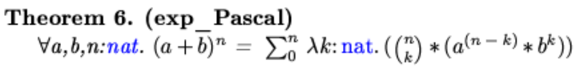
3.2 RSA Formalization

RSA is an asymmetric public key encryption algorithm which relies on prime number factorization. Its correctness has been formalized in various proof assistants including COQ by J.C. Almeida, and more recently by L. Théry. The complete formalization consists of half a dozen COQ scripts available at [?].

In Binomials.v, one finds the well known Pascal statement on binomals:

```
Theorem exp_Pascal:
  (a, b, n : nat)
   (power (plus a b) n) =
   (sum_nm
    O n
    [k : nat] (mult (binomial n k)
                (mult (power a (minus n k)) (power b k)))).
```

While the fragment does not contain any hint for presentation, the structure generated by `tmdoc` is rich enough to allow for user macro definitions so as to provide the following output.



Theorem 6. (exp_Pascal)
$\forall a,b,n{:}nat.\ (a+b)^n = \sum_0^n \lambda k{:}\,\text{nat}.\,(\binom{n}{k} * (a^{(n-k)} * b^k))$

Observe, however, that the sum does not use the index k as one would expect. This is going to be a slight enhancement in the next stage of our development. In short, this requires that macro definition can inspect the structure of its arguments, and in our case, grabs the k loop index to put it under the Σ symbol.

For a complete presentation of our contribution to this development, the required material is available from TMCOQ web site [?]. Besides the same documentation interface as for the standard library, our contribution offers also a mechanism for automatically generating an article from the set of files. This is based on a SCHEME script which is easily configurable.

4 Enhanced Features

One of our initial goals was to preserve the *invariant* that the script part of the document remains certifiable by the prover at any time. For this purpose, we have developed several functionalities described below, and based on the following remarks. In the introduction, we stated that a Coq script is a source program, where the inactive regions tagged as comments are good places for inserting documentation. The writing of an article from a formal development gives us another view. Everything is printable text, unless the macros or environments treat elements differently. In Knuth's view, *everything*, each character, could be active. However, for the casual user, the symbol '\' is commonly known as the first character of a document fragment which is going to be executed by the TeX engine. So the key idea is that the user is working in a *dual world* with respect to program source editing. Our tool transforms any *program source* file into a dual *document source* file, since TeX$_{\text{MACS}}$ is basically working in the same way as TeX.

Another property we get with the TeX$_{\text{MACS}}$ view of our document is that it is strongly *structured*. This structure allows us to introduce various interesting features including communication with XSLT oriented tools, Coq script recovery, LaTeX exportation, and interactive certification. This is made possible because we can deal with all these features using a single interpretation model. The key idea consists in traversing the document according to its structure. For each Coq related tag, we transform the corresponding subtree according to our specific needs.

4.1 A Portable XML Format for External Purposes

With respect to the structural shape of TeX$_{\text{MACS}}$ documents, it must be noted that TeX$_{\text{MACS}}$ does not yet use MathML standards for file format on disk (this is ongoing work by its development team). However, the XML format has been introduced recently to the set of import/export filters and, quite independently, by us for a particular purpose, explained thereafter.

In the context of the MoWGLI European project [?], there is a need for allowing annotations and reorganisation of proof fragments in a way that makes the presentation more readable. Since the proofs come from Coq and this feature requires an editor, our extension to TeX$_{\text{MACS}}$ was a good candidate.

XML is the dedicated format for documents inside the MoWGLI project, together with an intensive use of stylesheets (XSLT) as transformation language. Therefore, we have proposed a specific importation/exportation filter for XML. While we could have decided for a specific set of filters wrt to Tm-Coq, it is better offering a generic way of conversion because such documents should be allowed for edition even outside the editor, for instance by means of the edition of stylesheets. Therefore, we have defined the most general DTD (see figure **??**), which does not restrict to documents generated by tmdoc, but describes any TeX$_{\text{MACS}}$ document as made of several paragraphs, and enriched with special macros and environments, in a same way as we are used to with LaTeX documents.

Since we concluded our experiment, TeXmacs has also been enriched with such a format, although based on a different view, and consequently without a generic DTD. From both MoWGLI and TmCoq sides, our approach is robust enough for not requiring more than rewriting the outputs generated by templates (into stylesheets), owing to the common structural approach. Hence, we are going to provide the same functionality for TeXmacs users.

```
                    <!--    texmacs.dtd -- v 1.0      -->

        <!ENTITY % TeXmacs PUBLIC "-//TeXmacs//DTD TeXmacs Document//EN"
       "http://www-sop.inria.fr/lemme/Philippe.Audebaud/tmcoq/texmacs.dtd">

<!-- Document structure                                                -->

<!ELEMENT document      ( paragraph )+                                  >
<!ELEMENT paragraph     ( macro | sym | spc )*                          >

<!ELEMENT macro         ( document | paragraph )*                       >
<!ATTLIST macro         name CDATA #REQUIRED                            >

<!ELEMENT spc           EMPTY                                           >

 <!-- Takes care of specials like <rightarrow> symbol within text     -->

<!ELEMENT sym           EMPTY                                           >
<!ATTLIST sym           arg CDATA #REQUIRED                             >

<!-- Symbols                                                           -->

   <!ENTITY % HTMLlat1 PUBLIC "-//W3C//ENTITIES Latin 1 for XHTML//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml-lat1.ent"                       >

                              %HTMLlat1;
```

Fig. 4. Document Type Definition for the TeXmacs Documents

4.2 Coq *script (re)exportation*

In the introduction, we have explained the need for going back and forth between the Coq formal development and the document edited with TeXmacs. Hence the re-exportation towards a Coq script is required. As a matter of consequence, we need also keep the documentation (or informal part) into the exported script in order to recover this part afterwards, coming back to the documentation step.

Assume we are dealing with a TeXmacs document generated from a development. Among all the material present in this document, our tool ensures that any vernacular command is perfectly identified with the help of a specific macro (or environment), as already noticed. Thus, it is an easy task (thanks to TeXmacs's Guile facility) to recover in the correct order all the vernacular commands among the rest of Coq-irrelevant material within the document. Once collected and converted into the actual Coq concrete syntax, these pieces of text compose the Coq script's active part that is thus extracted from the TeXmacs document, as shown in figure **??**.

Everything else is considered as documentation. To some extent, this could be inserted as comments into the script, but comments are thrown away by the Coq lexical analyzer. In order to collect the documentation provided in the source script, we had to change the lexer behavior, so that not all the comments are ignored. However, doing this in every place where comments may occur would result in a parser with unreasonable complexity. We decided on a model where documentation and code alternate, but no documentation is looked for within the vernacular commands. Documentation fragments may appear in many places within the TeXmacs representation of a given script. We ensure that they do not get lost through the exportation step by placing them in places where the (modified) Coq lexer is able to find them. However, we cannot guarantee that their exact position is preserved.

4.3 LaTeX exportation

TeXmacs already supports LaTeX exportation. However, the documents we are dealing with are particular ones.

Therefore, we have split the exportation in two steps, where the first one profits from the fact that we are able to identify the fragments specific to Coq among the whole document. This first step is designed as a Guile script filter, which can be rewritten by the user at will. Selecting the exported material can be locally tuned inside the document (source) by assigning values to specific variables locally, following the same idea as marking a fragment as math, **bold,** or *italic*. Afterwards, everything is put in the hands of the common LaTeX exportation filter. This solution offers both portability and enough expressiveness from the user point of view.

4.4 Coq-TeXmacs

Let us have a look at the `coq-tex` tool provided with Coq distribution. This is a stand-alone program which processes Coq phrases embedded in LaTeX files. This tool *greps* the LaTeX for Coq vernacular commands, identified by particular LaTeX environment tags (`coq_examples`, `coq_examples*`, `coq_eval`). The result is stored in a temporary script and sent to Coq toplevel (`coqtop`) for evaluation. Depending on the surrounding tag, the command itself or the result are then re-inserted in place in the LaTeX file. As it stands, the environment tags offered with `coq-tex` gives the user the different possibilities based on two criteria: i) is the vernacular command to be displayed and ii) is the result of its evaluation to be displayed? Whenever the answer to a question is positive, the appropriate text is printed verbatim at the appropriate place.

Assuming instead we start with a TeXmacs file, we can simulate the same behavior. However, we can do much more! The displayed material is no longer restricted to the raw verbatim format, nor the simple combination of two criteria. Moreover, since the source document is structured, there is no need for preparing an intermediate Coq script; whenever required, the corresponding answers can be directly inserted following the command which has been sent. Thus, our coq-texmacs implements this functionality, based on the above exportation tools.

4.5 Certification in edition mode

In fact, we are very close to provide another functionality which we are going to explain by comparing it with a spelling checker. Inside TeXmacs the spelling checker is launched the same way as inside Emacs. That is, the editor selects each word one after the other and sends it to the spelling checker. The spelling checker is launched in background. As far as spelling is concerned, the (atomic) item that has to be isolated in the buffer is a word, meaning a sequence of characters delimited by spaces or punctuation marks, depending on the underlying language. Assume now the spelling checker is Coq, words are to be replaced by vernacular commands. But we know how to isolate this kind of item, therefore making TeXmacs smart enough to send each vernacular command in the correct order to some Coq process run in background. This mode offers a way to *certify* the (underlying) formal development inside the editor. Obviously, the expected behavior cannot be the same as for a spelling checker. Wherever an error is encountered, this might inhibit the certification mechanism to continue. Actually, some dependency information between statements and their proofs will prove sufficient to allow to have a partial run of the certification process, leaving rejected (subtrees) pieces of our development marked as requiring further correction.

4.6 Interactivity

Since the user is working inside an interactive editor, it is highly desirable to offer on the fly certification, which requires that the editor keeps communicating with a Coq process. Towards this goal we have experimented with TeXmacs interactive mode with Coq. In section **??**, we have shown that the editor TeXmacs offers a session mode with external programs. To make such communication possible with Coq, very little is needed as the following session with Coq shows.

A sample session is presented in figure **??**. From this example, we can make some useful observations. Up to now, our explanations were mainly concerned with formulas. Actually, exporting a full script does not raise any more difficult issues. But when interacting with Coq, it appears that the prover really decides the presentation in various contexts! The information displayed during a proof session, error messages, or answers to vernacular

```
          Welcome to Coq 7.4 (Feb 2003)
                Interactive Session
Coq < Print all

   all  =  λA:Set; P:A
   → Prop. ∀x:A. (P  x)  :  ∀A:Set. (A  → Prop) → Prop

Coq < Lemma 1. (foo)
      ∀a,b:Prop. a  → b  → a ∧ b

   ________________________
   ∀a, b: Prop. a → b → a ∧ b

foo < Intros.

    a   :  Prop
    b   :  Prop
    H   :  a
    H0  :  b
   ____________
       a ∧ b

foo < Split.

   2 sub-goal(s)

    ⋮
   __
   a

   Subgoal 2 is b

foo < Exact H

    ⋮
   __
   b

foo < Trivial

   Subtree proved

foo < Save

   foo is defined

Coq < Print f

   Error : [6 - 7]  Error: f not a defined object

Coq < Print foo

   foo  =  λa,b:Prop; H:a; H0:b.
    <a, b> {H, H0}  :  ∀a,b:Prop. a  → b  → a ∧ b

Coq < Print bool

   Inductive 2. (bool) : Set :=
    | true : bool
    | false : bool
```

Fig. 5. A sample session with TmCoq

commands are controlled by the prover.

The most interesting issue concerns proof editing. Along the lines of the above session, one would expect the editor to allow the user to edit the proof script, rather than accumulating steps in the buffer. This is possible in $\mathrm{T_EX_{MACS}}$ thanks to the Scheme extension language and provided through specific tree commands, together with the possibility for the document to be modified by the running program (as well as the user or the editor interface).

5 Conclusion and Perspectives

Our initial motivation was to offer the users more powerful solutions for documenting their Coq formal developments. Existing tools already offer useful means but lack the capability to render such mathematical or logical development in the usual language for scientific documents and none of them permit significant modifications or even final adjustments while preserving the fundamental property that the underlying formal document has been certified by the prover. In the first case, we conclude that this is only possible at the cost of working behind a dedicated parser. Due to complexity of the language dealt with by Coq and also motivated by the purpose to share our work and offer Coq various output formats, we decided to implement our documentation generator tmdoc as part of the Coq source distribution. At this stage, the rendering of mathematical formulas and the possibility to prepare a scientific article presenting the formal development was within the capabilities of the scientific editor $\mathrm{T_EX_{MACS}}$.

The fact that this editor supports the Guile/Scheme extension language has also made it possible to handle the second case beyond our initial expectations. We provide Coq and LaTeX exportation functions both through the editor interface and as stand-alone scripts, which offer at the same a certification mechanism (on the Coq script) and the article production in the format commonly accepted by the scientific community.

Beyond these tools, our experimentation has shown very attractive perspectives. We want to achieve the functionality allowing the user to interact with the prover in a mode as transparent as possible. That means that the user will be allowed to write definitions, lemmas, and the other mathematical material in the syntax offered by the editor, leaving to the system all the conversions to be done in order to match the actual concrete syntax understood by the prover. Our view is that $\mathrm{T_EX_{MACS}}$ offers a nice interface for abstraction with respect to the prover. The interface can be adapted so that the user's syntax and commands do not depend on the prover. As such, this interface should offer a convenient support to teach beginners (students for instance) how to develop formal proofs. Even the certification mechanism could allow for machine-assisted verification of students' examinations. We consider our current experiments as an important step in this direction.

References

[1] *Isabelle: Theorem Proving Environment.* http://isabelle.in.tum.de/ and http://www.cl.cam.ac.uk/Research/HVG/Isabelle/.

[2] *The Maple analytical computation software.* http://www.maplesoft.com/.

[3] *Mathematica: The Way the World Calculates.* http://www.wolfram.com/products/mathematica/.

[4] Philippe Audebaud and Laurence Rideau. *Coq integration within TEXMACS.* Home page: http://www-sop.inria.fr/lemme/Philippe.Audebaud/tmcoq/.

[5] Jean-Christophe Filliâtre. *Documentation tool for Coq.* Available at http://www.lri.fr/~filliatr/coqdoc/.

[6] SRI Computer Science Laboratory. *The PVS Specification and Verification System.* http://pvs.csl.sri.com/.

[7] Lemme Team, INRIA Sophia-Antipolis. *Pcoq, A graphical user-interface for Coq.* Home page: http://www-sop.inria.fr/lemme/pcoq/.

[8] Cornell Prl Automated Reasoning Project. *MetaPRL.* http://cvs.metaprl.org:12000/metaprl/default.html.

[9] Coq Team. *The Coq Theorem Prover Assistant.* Available at http://coq.inria.fr/.

[10] Laurent Théry. *A proof of correctness of RSA algorithm that relies on Fermat's little theorem.* Browsable online at ftp://ftp-sop.inria.fr/lemme/Laurent.Thery/Rsa/index.html.

[11] MoWGLI*: Mathematics on the Web: Get it by Logic and Interfaces.*(European IST) Home page: http://www.mowgli.cs.unibo.it/.

[12] Philippe Audebaud and Luca Padovani. *(D4.c) Prototype functionalities for assisted annotation.* In MoWGLI deliverables http://mowgli.cs.unibo.it/misc/deliverables/interfaces/

[13] Joris van der Hoeven. *The GNU TeXmacs free scientific text editor.* Available at http://www.texmacs.org/.