Integrating Proof Assistants as Reasoning and Verification Tools into a Scientific WYSIWYG Editor

Serge Autexier Christoph Benzmüller Armin Fiedler
Henri Lesourd

Saarland University, Department of Computer Science,
P.O. Box 15 11 50, D-66041 Saarbrücken, Germany

{serge,chris,afiedler,henri}@ags.uni-sb.de

Abstract

A major problem for the acceptance of mathematical proof assistance systems in mathematical practise is the shortcomings of their user interfaces. Often the interfaces are developed bottom-up starting from the mathematical proof assistance system. Therefore they usually focus on the individual system and its proof development paradigm and neglect traditional forms to communicate proofs as used by mathematicians. To address this problem we propose a top-down approach where we start from an existing scientific WYSIWYG text editor which supports the preparation of mathematical publications in high quality typesetting and integrate a mathematical proof assistance system to support proof development and validation. Concretely, we extend the document format of the text editor by semantic markup to encode formal mathematical content and to communicate with the formal system. Additionally we provide interaction markup defining context-sensitive means to control the mathematical proof assistance system through the text editor.

1 Introduction

Unlike computer algebra systems (CASs), mathematical proof assistance systems have not yet achieved considerable recognition and relevance in mathematical practise. Clearly, the functionalities and strengths of these systems are generally not sufficiently developed to attract mathematicians on the edge of research. For applications in e-learning and engineering contexts their capabilities are often sufficient, though. However, even for these applications significant progress is still required, in particular with respect to the user-friendliness of the interfaces of these systems. One significant shortcoming of the current systems is that they are not fully integrated or accessible from

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

within standard mathematical text editors and that therefore a duplication of the representation effort is typically required. For purposes such as tutoring, communication, or publication, the mathematical content is in practise usually encoded using common mathematical representation languages by employing standard mathematical editors (e.g., LaTeX and Emacs). Proof assistants, in contrast, require fully formal representations and they are not yet sufficiently linked with these standard mathematical text editors.

Therefore, rather than developing a new user interface for the mathematical assistance system ΩMEGA [15], the approach suggested in this paper is to adapt ΩMEGA to serve as a mathematical service provider for the scientific text editor TeXmacs [7]. Thereby, the main difference is that instead of providing a user interface only for the existing interaction means of ΩMEGA, we extend ΩMEGA to support requirements as they arise in the preparation of a mathematical document.

Generally, we aim at an approach that is independent of the particular proof assistant system to be integrated. We present several aspects of this more general viewpoint, but the focus of this paper is on the direct link between TeXmacs and ΩMEGA.

TeXmacs [7] is a scientific WYSIWYG text editor that provides professional typesetting and supports authoring with powerful macro definition facilities like in LaTeX. It moreover allows for the definition of plug-ins that automatically process the document and is thus especially well suited for an integration with a mathematical assistance system.

The mathematical proof assistance system ΩMEGA [15] is based on the Core calculus [2], which supports proof development directly at the *assertion level* [9], where proof steps are justified in terms of applications of definitions, lemmas, theorems, or hypotheses (collectively called *assertions*). In particular, ΩMEGA allows for proof development at a high level of abstraction using knowledge-based proof planning. The proofs can then be verbalised in natural language via the proof explanation system *P.rex* [8].

Now, consider a teacher, student, engineer, or mathematician who is about to write a new mathematical document in TeXmacs. A first crucial step in our approach is to link this new document to one or more mathematical theories provided in a mathematical knowledge repository. By providing such a link the document is initialised and TeXmacs macros for the relevant mathematical symbols are automatically imported; these macros overload the pure syntactical symbols and link them to formal semantics. In a TeXmacs display mode, where this additional semantic information is hidden, the user may then proceed with editing mathematical text as usual. The definitions, lemmas, theorems and especially their proofs give rise to extensions of the original theory and the writing of some proof goes along with an interactive proof construction in ΩMEGA. We define TeXmacs macros to annotate theory-specific knowledge such as types, constants, definitions and lemmas. This allows us to translate new textual definitions and lemmas into the formal representation,

as well as to translate (partial) textbook proofs into formal (partial) proof plans. Thus, the semantic annotations are used to *automatically* build up a corresponding formal representation in ΩMEGA, thus avoiding a duplicated encoding effort of the mathematical content. Altogether, this allows for the development of mathematical documents in professional type-setting quality which in addition can be formally validated by ΩMEGA, hence obtaining *verified mathematical documents*.

This paper is organised as follows: First, we describe our approach in abstract terms in Section 2 and give an overview of TEXMACS in Section 3. Next, in Section 4, we show how we represent mathematical documents with semantic markup in TEXMACS. Then, we add further markup that allows the user to interact with the proof assistant through TEXMACS in Section 5 and we present an example in Section 6, before we discuss related work and conclude the paper.

2 Our Approach

In our approach mathematical documents are prepared within TEXMACS in direct interaction with ΩMEGA. This requires that (1) the semantic content of the document is accessible for formal analysis and (2) the interactions in either direction should be localised and aware of the surrounding context.

Making the document accessible for a formal analysis requires the extraction of the semantic content and its encoding in some representation suitable for further formal processing. Instead of automatically extracting the semantic content we currently use semantic annotations in the TEXMACS document, which must be provided by the author. Therefore, one requirement is to keep the burden of providing these annotations as low as possible.

Formal representations of mathematical objects in existing mathematical assistance systems such as definitions or proofs are very detailed whereas many formally necessary details that are considered trivial or easily inferable are omitted in documents written by human beings. Thus, there is a big gap between common mathematical language and formal, proof-system-oriented representations. Hence, another requirement to interfacing TEXMACS to ΩMEGA is to keep the amount of detail that must be provided by the user in the TEXMACS document at an acceptable level.

We expect that the preparation of a mathematical document in interaction *with* a proof assistance system requires more effort from the authors than without, since the authors have to provide the semantic annotations for the different elements in the document, for instance. They will accept the additional burden only if they receive an added value in return. Not speaking of the benefits of semantics-based management and improved search support, an immediate benefit of preparing a document in interaction with a proof assistance system is the increase of the quality of the paper in terms of reliability.

These advantages are achieved by specific functionalities provided by the

proof assistance system, such as the ability

- to determine available symbols, definitions, axioms and lemmas known in some specific context inside the document structure,
- to perform type-checking of arbitrary parts of the document, that is, of the semantic elements contained therein with respect to the context determined from the surrounding document parts, and
- to provide sophisticated proof-checking facilities that go beyond simple checking of calculus rule applications.

This last feature not only increases the reliability of the document, but actually can save the author time by automatically proving subgoals. Using existing techniques to generate natural language verbalisations for formal proofs [9,8] these textual proof fragments can even be directly integrated into the document.

In order to allow both the user and the proof assistance system to manipulate the mathematical content of the document we need a common representation format for this pure mathematical content implemented both in TeXmacs and in ΩMEGA. To this end we define a language S, which includes many standard notions known from other specification languages, such as terms, formulae, symbol declarations, definitions, lemmas and theorems. The difference to standard specification languages is that our language S

- includes a language for proofs,
- provides means to indicate the logical context of different parts of a document, which adjust to the textual structure of documents and are less rigid than the usual structured theories used in specification languages, and
- accommodates various aspects of underspecification, that is, formal details that the writer purposely omitted.

Given the language S, we augment the document format of TeXmacs by the language S. Thus, if we denote the document format of TeXmacs by T, we define a *semantic* document format $T + S$. By $T + S$ we mean a conservative, that is, still TeXmacs-acceptable, extension of the language T by language S. Note that especially a text-fragment written in S only is an acceptable TeXmacs-text fragment and there is projection π from $T + S$ to S.

Ideally, this format of the documents and especially the semantic annotations should not be specific to ΩMEGA in order to allow for the combination of the TeXmacs extension with other proof assistance systems as well as the development of independent proof checking tools. However, an abstract language for proofs that is suitable for our purposes and that allows for underspecification is not yet completely fixed and thus our implementation currently supports only the assertion-level proof construction rules provided by Core [2]. Hence, instead of defining a fixed language S, we divide S in one part addressing mathematical concepts, formulae, and so forth, and one part

for mathematical proofs. The aim is to replace the current Core-dependent proof representation language by the general purpose abstract proof language under development in due time. Our current choice for S is presented in Section 4. It supports the *static* representation of semantically annotated documents, which can be professionally typeset with TEXMACS.

Note that the mathematical content of the document is represented both in TEXMACS (in the language $T + S$) and in the proof assistant (in the language S). Since both the user and the proof assistant must be able to manipulate the mathematical content, both representations (namely in $T + S$ and in S) must be synchronised. This synchronisation is done using a diff/patch mechanism tailored to the tree structure of the TEXMACS documents.

Consider two versions ts_i and ts_{i+1} of the document represented in $T + S$ and their counterparts s_i and s_{i+1} in S. When the user edits the document, thus transforming it from ts_i and ts_{i+1}, the differences between ts_i and ts_{i+1} are compiled into a patch description p, which applied to s_i yields s_{i+1}. For the opposite direction, assume the proof assistant works on s_j obtained from the TEXMACS document ts_j by the projection π. If the proof assistant changes the document version s_j into s_{j+1}, the patch description p' between the two versions is also a patch description from ts_j to ts_{j+1}, since

- deletion of a subdocument d_s in s_j corresponds to the deletion of the subdocument d_{ts} inside ts_j, where $\pi(d_{ts}) = d_s$;
- insertion of a subdocument d_s at some position into s_j corresponds to the insertion of d_s at the corresponding position inside ts_j. Note that this is possible since S is a sublanguage of $T + S$, and hence any text-fragment in S is a valid $T + S$-text fragment.

Hence, the patch description p is used to propagate any changes the user made to the proof assistant, and p' is used to update the TEXMACS document whenever the proof assistant modifies the mathematical content.

In order to translate the patch descriptions—that is, the positions where text-fragments are inserted or deleted—a key-based protocol is used to identify the corresponding parts in $T + S$ and S. The language for keys is presented in Section 4 and the synchronisation mechanism is described in Section 5.

Beyond this basic synchronisation mechanism, we also present in Section 5 a language that allows for the description of specific interactions between TEXMACS and the proof assistant. This language M is a language for structured menus and actions with an evaluation semantics that allows the user interface to flexibly compute the necessary parameters for the commands and directives employed in interaction with the proof assistants. The TEXMACS document format $T+S$ from Section 4 is finally extended to $T+S+M$, where the menus can be attached to arbitrary parts of a document and the changes of the documents performed either by the author or by the proof assistants are propagated between $T+S+M$ and $S+M$ via the diff/patch mechanism. Note that this includes also the adaptation of the menus, which is a necessary

prerequisite to support context-sensitive menus and actions contained therein.

3 TEXmacs: A Scientific WYSIWYG Editor

TEXMACS is a WYSIWYG editor that allows for easy authoring of mathematical documents. These mathematical documents can be stored as ASCII files annotated with *markup*. For example, the following sum

$$2\sum_{i=1}^{n} a_i \int_a^b f_i(x)g_i(x)\,dx$$

is encoded in TEXMACS markup as

```
<\equation*>
  2<big|sum><rsup|n><rsub|i=1>a<rsub|i><big|int><rsup|b><rsub|a>
  f<rsub|i>(x)g<rsub|i>(x) d<no-break>x
</equation*>
```

Once loaded into TEXMACS, a document is internally represented as a *tree*. Such a tree can be displayed in different ways, such as in the markup language, in LaTeX syntax or as a Lisp s-expression.

The architecture of TEXMACS is depicted in Figure 1. The two components that are most important for our purposes are the macro processor and the event processor.

The *macro processor* expands the user-defined macro markup tags occurring in a tree and translates them into primitive, macro-free TEXMACS markup. One can define new macro tags and the corresponding TEXMACS macros to determine the *pretty-printing* of these new tags. Once expanded, the rewritten tree is displayed in the current output device (e.g., the screen, a PostScript file, or a PDF file) by the *typesetter*.

The *event processor* detects the user input events and updates the current markup tree accordingly. Once updated, the current tree is then expanded and again typeset. One can define new *user inputs* (e.g., keyboard shortcuts, mouse events), and write a TEXMACS *plug-in* to define the corresponding new *actions* that are to be performed on the current markup tree in the editor. Such plug-ins are extensions to the event processor, and can be written using the *Scheme* scripting language, which is embedded in TEXMACS. They can also be employed to allow external systems such as CASs or theorem provers to modify the current markup tree.

Markup and Markup Syntax

Let us consider a basic example. The TEXMACS markup fragment

```
<with|font-shape|italic|<underline|Hello!>>
```

for instance, is typeset as *Hello!*

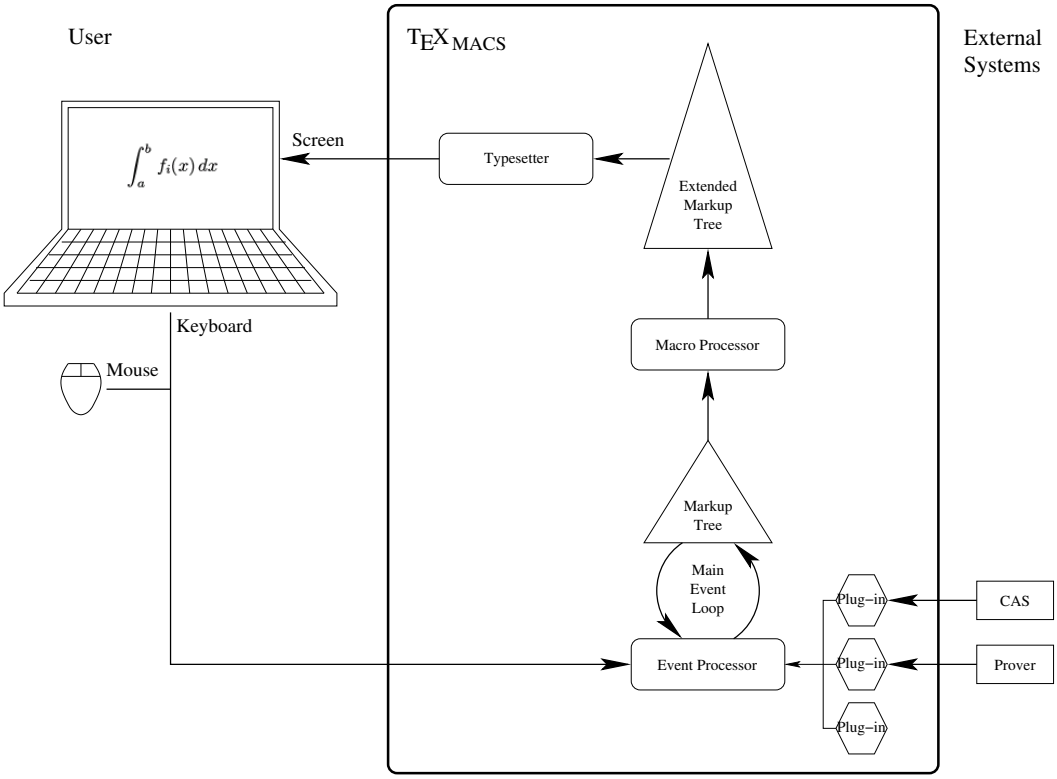


Fig. 1. The TEXMACS architecture.

Recall that the markup that one can find in a TEXMACS document file is only a *description* of the markup tree represented internally. If we use the *Scheme* syntax instead, the markup fragment above is written-printed as the s-expression

```
(with font-shape italic (underline "Hello!"))
```

Macros

In this section, we show a very basic example of the TEXMACS macro language (cf. [17] for more details). Here is a simple TEXMACS macro that can be used to turn a part of the document into *italics underlined* text:

```
(underlined-italics x)
  ⟹ (with font-shape italic (underline (arg x)))
```

The left-hand side of this expression stands for the use of the macro and the right-hand side for its expansion. Given this definition of the macro `underlined-italics`, the markup fragment

```
(underlined-italics "This is italics underlined text.")
```

is first rewritten by the macro processor as

```
(with font-shape italic
  (underline "This is italics underlined text."))
```

and then displayed in TEXMACS as :

This is italics underlined text.

Processing the Markup Using *Scheme*

The event processor can be extended by plug-ins defined in *Scheme* scripts. These scripts can manipulate the internal markup tree that represents the document, for example, by inserting new macros, or by dynamically defining new macros.

4 Formal Representation of Mathematical Documents

For illustration purposes we use a simple example theorem throughout this paper, one of the deMorgan theorems: For any two sets A and B we have

$$\overline{A \cup B} = \overline{A} \cap \overline{B}$$

where $\overline{X}$ denotes the complement of a set X. Using this theorem we illustrate the different aspects of its processing inside our system. First, the mathematical knowledge is encoded and stored into well-defined parts of a given TEXMACS document. These well-defined parts are the language S from Section 2 and are called *mathematical fragments*. Mathematical fragments can be surrounded by the usual markup (the language T from Section 2) that authors use to write mathematics in natural language in a document.

Currently, the different kinds of mathematical fragments we use are:

- symbols, declarations and terms
- definitions and theorems
- proofs and proof steps
- keys and links
- contexts and environments

4.1 Basic Encoding of Mathematical Fragments

We now present the grammar for the language S used to encode the semantics of mathematical documents in TEXMACS. Since the language S extends the language T of TEXMACS documents, we use the non-terminal symbol `ELEMENT` to denote the language T and describe the extension by incrementally including the concepts of S in T. For the sake of clarity, however, we give only simplified versions of the grammar rules here. In the following, we assume the non-terminals `SYMBOL` and `STRING`, that can be seen as symbols and strings, respectively, in the sense of Lisp. The symbol '_' is a terminal symbol for unspecified names.

```
NAME        ::= SYMBOL | _
DECLARATION ::= (type-constant NAME) | (type-variable NAME)
                | NAME
TERM        ::= SYMBOL | (SYMBOL TERM+) | (BINDER NAME TYPE TERM)
BINDER      ::= forall | exists | lambda
TYPE        ::= TERM
```

Note that the symbols occurring inside a `TERM` should have been previously defined in a `DECLARATION` construct. Finally, we extend the language T by adding the rule

```
ELEMENT     ::= ELEMENT | DECLARATION | TERM
```

4.1.1 Typesetting Mathematical Fragments

Given our example formula $\overline{A \cup B} = \overline{A} \cap \overline{B}$, we can define the following macros for the operators needed:

```
(set-equal a b)
  ⟹ (with mode math (arg a)‘‘=’’(arg b))    displayed as   a = b
(set-union a b)
  ⟹ (with mode math (arg a)<cup>(arg b))    displayed as   a ∪ b
(set-intersection a b)
  ⟹ (with mode math (arg a)<cap>(arg b))    displayed as   a ∩ b
(set-complement a)
  ⟹ (with mode math (overline (arg a)))     displayed as   ā
```

Based on these macros, the markup fragment

```
(set-equal (set-complement (set-union A B))
           (set-intersection (set-complement A)
                             (set-complement B)))
```

is displayed as

$$\overline{A \cup B} = \overline{A} \cap \overline{B}$$

Thus, TEXMACS macros provide us with a very elegant and very simple way to define an encoding and — at the same time — the corresponding typesetting for a given set of mathematical operators. Further mathematical concepts such as $\in$ or $\Leftrightarrow$ can be similarly encoded by macros; we omit their definitions in this paper.

Notation

In order to provide a better readable way to present a long s-expression in this paper, we write parts of this s-expression as they are displayed in the TEXMACS editor embraced by *double angle brackets* ('<<', '>>'). Our previous markup fragment

```
(set-equal (set-complement (set-union A B))
           (set-intersection (set-complement A)
                             (set-complement B)))
```

could for example also be written as

(set-equal <<$\overline{A \cup B}$>> <<$\overline{A} \cap \overline{B}$>>)

4.1.2 Encoding of the deMorgan Theorem

Now, we present the grammar for assertions, that is, theorems, definitions and assumptions:

```
THEOREM     ::= (theorem    NAME DECLARATION* ASSUMPTION*
                            CONCLUSION)
CONCLUSION  ::= (conclusion NAME TERM)
ASSUMPTION  ::= (assumption NAME TERM)
DEFINITION  ::= (definition DECLARATION* CONCEPT TERM)
CONCEPT     ::= (type-constant NAME) | (constant NAME TYPE)
ELEMENT     ::= ELEMENT | DEFINITION | THEOREM
```

Then, the conclusion of the deMorgan theorem can in this language be encoded as

```
(conclusion  _
 (forall A set (forall B set
 (set-equal (set-complement (set-union A B))
            (set-intersection (set-complement A)
                              (set-complement B))))))
```

and the whole theorem can then be encoded by

(theorem deMorgan (conclusion _ <<$\forall A : \mathrm{set} \,.\, \forall B : \mathrm{set} \,.\, \overline{A \cup B} = \overline{A} \cap \overline{B}$>>))

Given the appropriate set of macros for all the occurring symbols, this markup is displayed in the TEXMACS editor as

Theorem *deMorgan*:
Conclusion:
$\forall A : \mathrm{set} \,.\, \forall B : \mathrm{set} \,.\, \overline{A \cup B} = \overline{A} \cap \overline{B}$

4.2 Proofs and Proof Steps

We now define the markup grammar for proofs:

```
PROOF                 ::= (proof @THEOREM PROOF-STEP)
PROOF-STEP            ::= (proof-step CONTENT RULE-DESCRIPTION
                                      (PROOF-STEP*) MENU)
RULE-DESCRIPTION      ::= open | closed | (apply TERM TERM)
                        | (focus TERM+) | (unfocus) | ...
CONTENT               ::= (GOAL AVAILABLE-ASSUMPTIONS
                                ALTERNATIVE-GOALS)
ALTERNATIVE-GOALS     ::= (TERM*)
AVAILABLE-ASSUMPTIONS ::= (TERM*)
GOAL                  ::= TERM
ELEMENT               ::= ELEMENT | PROOF-STEP | PROOF
```

where `@THEOREM` describes a slot which must contain a link to a fragment of the type `THEOREM`. Note that the non-terminal `MENU` is used to encode the context-sensitive menu which is available for the respective proof-step. The detailed definition of the language for menus is given in Section 5.

To illustrate the proof markup language, we show a (slightly simplified) encoding for the first two proof steps of our deMorgan theorem, after applying the rule stemming from the assumption $\forall A : \text{set} \,.\, \forall B : \text{set} \,.\, A = B \Leftrightarrow \forall x : \text{element} \,.\, x \in A \Leftrightarrow x \in B$:

```
(proof @THEO1
  (proof-step
    (<<∀A : set . ∀B : set . A ∪ B = A ∩ B>> (...) (...))
    (apply <<A = B ⇒ ∀x : element . x ∈ A ⇔ x ∈ B>>)
    ((proof-step
       (<<∀A : set . ∀B : set . ∀x : element . x ∈ A ∪ B ⇔ x ∈ A ∩ B>> (...)
                                                                     (...))
       open
       ()
       (menu ...)))
    (menu ...)))
```

4.3 Keys and Links

In order to allow actions in menus to refer to parts of a TEXMACS document as well as to design localised patch applications we introduce a system of symbolic *keys* and *links* to denote parts of the document. Since these keys and links will have a specific semantics, we keep them separated from the standard label and reference mechanisms as, for instance, known from LaTeX and also available in TEXMACS.

To this end, we decided to define a general way to attach *attributes* to the fragments. In particular, important attributes are the *key*, which can be seen as an identifier, and the *link*, which refers to the identifier of a different fragment. We denote elements for which a key has been defined as *keyed elements*. The links can either directly point to these keyed elements or to a sub-element of a given keyed element by additionally providing the *path* inside this element. Keyed elements may also contain further attributes. We define:

```
KEYED-ELEMENT ::= (keyed (ATTR*) ELEMENT)
ATTR          ::= (SYMBOL ELEMENT)
ELEMENT       ::= ELEMENT | KEYED-ELEMENT
```

For example, the formula $1 + 2 \times 3$ with the key `FORM1` and attributes for the depth and the number of leaves can be encoded as an s-expression in TEXMACS as follows:

```
(keyed ((key FORM1)(depth 2)(leaves 3))
       (expr + 1 (expr * 2 3)))
```

Notation

For the sake of readability, we introduce the following notation: We use *curly braces* ('{', '}') for the attributes, and the attributes themselves are written with '=' as separator between the name and the value. The previous formula is thus presented as

```
{ key=FORM1 depth=2 leaves=3 } (expr + 1 (expr * 2 3))
```

The *links* referring to keyed elements are defined as follows:

```
LINK    ::= (link KEY PATH)
PATH    ::= (NAT*)
ELEMENT ::= ELEMENT | LINK
```

Notation

In this paper we also write a link `(link KEY PATH)` as `@KEY(PATH)`. For example, a link to the `(expr * 2 3)` subexpression in the formula `FORM1` will be written as `@FORM1(2)`, where `(2)` is the path referring to the second argument term inside `(expr + 1 (expr * 2 3))`.

4.4 Contexts and Environments

Generally, the mathematical fragments one can define in a document may use symbols that refer to other objects (such as definitions or theorems) that are not defined in the current document. These external objects "live" in packages that we call *theories*. These theories could be stored in other TEXMACS documents or in a knowledge base. To indicate that we write a fragment while being in the name space of a given theory (or, as we say, in the *context* of the theory), we introduce a syntactic construct in the TEXMACS markup, called `context`:

```
CONTEXT ::= (context THEORY BODY)
THEORY  ::= (theory-by-name NAME) | (theory-union THEORY+)
ELEMENT ::= ELEMENT | CONTEXT
```

Then, we can build nested markup structures using the `context` markup construct. Inside the `BODY` parts of one or several nested `context` markup constructs, one has access to the joint name spaces of all the theories which appear in the `THEORY` part of all the nested `context` markup constructs and also to all assertions collected from the root of the document downwards to the current position. The set of mathematical objects that are accessible from one position inside such a structure of nested contexts is called the *environment* at that position.

In the following, we assume that a theory `set-theory` is available providing definitions and lemmas in naive set theory:

$$\forall A : \text{set} \,.\, \forall B : \text{set} \,.\, A = B \Leftrightarrow \forall x : \text{element} \,.\, x \in A \Leftrightarrow x \in B \qquad (\textit{hyp1})$$

$$\forall A : \text{set} \,.\, \forall x : \text{element} \,.\, x \in \overline{A} \Leftrightarrow \neg(x \in A) \qquad (hyp2)$$
$$\forall A : \text{set} \,.\, \forall B : \text{set} \,.\, \forall x : \text{element} \,.\, x \in A \cup B \Leftrightarrow x \in A \vee x \in B \qquad (hyp3)$$
$$\forall A : \text{set} \,.\, \forall B : \text{set} \,.\, \forall x : \text{element} \,.\, x \in A \cap B \Leftrightarrow x \in A \wedge x \in B \qquad (hyp4)$$

Thus, the markup shown previously for the deMorgan theorem, for example, can be embedded into a `context` markup construct as follows:

```
(context (theory-by-name set-theory)
  { key=THE01 }
  (theorem deMorgan
    (conclusion _ <<∀ A : set . ∀ B : set . A∪B = A∩B>>)))
```

We observe the following:

- Given a theory stored in some mathematical repository, we need the set of TEXMACS macros corresponding to the symbols defined in it, so that we can display the symbols and terms in their common mathematical form in the TEXMACS editor. We implemented such a mechanism between ΩMEGA and the mathematical database MBASE [10], which contains the available theories together with their macro definitions. Given the name of a theory, the relevant macros can be loaded into TEXMACS from MBASE via ΩMEGA.
- Given a particular position inside a nested context structure, we can derive its environment, that is, the types, symbols, definitions, lemmas, theorems, etc. When the user begins a proof of a theorem, the environment of the theorem is sent to the proof assistant, in order to enable the prover to access the relevant concepts. Note that the environment not only contains the predefined elements imported from the database, but also the mathematical elements defined in the document, that is, symbols, definitions or lemmas newly introduced in the document. [1]

5 Synchronising Editor and Proof Assistants

In this section, we first describe how CORE, the logic layer of ΩMEGA, supports assertion application via conditional rewriting (cf. [2] for more details). We then describe our TEXMACS interface to CORE.

5.1 Basic Rewrite Steps in CORE

For each open proof line, CORE provides the goal and the list of available assertions. The application of one of these to a sub-formula of either the goal (backward reasoning) or another assertion (forward reasoning), requires

(i) to select a sub-formula of the former, in order to give the information about *how* to do the rewriting, and

[1] So far, capturing and uploading environments into the proof assistant has not yet been implemented. Currently, we encode the concepts needed (namely the types and the operations, and the hypotheses, which are encoded as assumptions of the theorem) inside the theorem.

(ii) to select a sub-formula of the latter in order to give the information about *where* the rewriting should take place.

For example, given the hypotheses shown previously, in the first proof step of our deMorgan proof example, the set of available assertions is:

$$\forall A : \text{set} \,.\, \forall B : \text{set} \,.\, \overline{A \cup B} = \overline{A} \cap \overline{B} \qquad (goal)$$
$$\forall A : \text{set} \,.\, \forall B : \text{set} \,.\, A = B \Leftrightarrow \forall x : \text{element} \,.\, x \in A \Leftrightarrow x \in B \qquad (hyp1)$$
$$\forall A : \text{set} \,.\, \forall x : \text{element} \,.\, x \in \overline{A} \Leftrightarrow \neg(x \in A) \qquad (hyp2)$$
$$\forall A : \text{set} \,.\, \forall B : \text{set} \,.\, \forall x : \text{element} \,.\, x \in A \cup B \Leftrightarrow x \in A \vee x \in B \qquad (hyp3)$$
$$\forall A : \text{set} \,.\, \forall B : \text{set} \,.\, \forall x : \text{element} \,.\, x \in A \cap B \Leftrightarrow x \in A \wedge x \in B \qquad (hyp4)$$

We decide to apply the definition of set equality (assumption *hyp1*) using the sub-formula "$A = B$" as the left-hand side of our rule, which we indicate by a box in this paper [2]:

$$\forall A : \text{set} \,.\, \forall B : \text{set} \,.\, \boxed{A = B} \Leftrightarrow \forall x : \text{element} \,.\, x \in A \Leftrightarrow x \in B$$

Using this selection, Core provides us with the following possible rewrite rule:

$$A = B \quad \longrightarrow \quad \forall x : \text{element} \,.\, x \in A \Leftrightarrow x \in B$$

In order to apply this rule to the relevant part of the goal, we can thus make the following selection inside the goal:

$$\forall A : \text{set} \,.\, \forall B : \text{set} \,.\, \boxed{\overline{A \cup B} = \overline{A} \cap \overline{B}}$$

As a result of the rewrite step, Core creates a new proof step, inside which the goal now is:

$$\forall A : \text{set} \,.\, \forall B : \text{set} \,.\, \forall x : \text{element} \,.\, x \in \overline{A \cup B} \Leftrightarrow x \in \overline{A} \cap \overline{B}$$

5.2 User Input and Evaluation

From the previous description, it follows that a user interface to Core must enable the user to access the list of the assertions for each proof step, and also allow for selecting the relevant sub-formulae in this list of assertions. More generally, the user interface should enable the user to provide the missing information that is needed to apply a theorem proving operation to the current proof step. In our user interface, the possible *user interface actions* can either be *apply* actions, which refer to some theorem proving operation, or be *choice* actions (or *choices*), which provide a way for the user to select a subset out of a given dataset (e.g., a list of formulae). The sub-language to define these menus is given by the grammar rules starting from the non-terminal symbol `MENU`:

[2] Note that TEXMACS implements the proof-by-pointing approach [6], where the selection is done by pointing to a sub-formula with the mouse.

```
MENU          ::= (menu EVALUATE ACTION)
EVALUATE      ::= BOOLEAN
ACTION        ::= APPLY | CHOICE
APPLY         ::= (apply  LABEL FOLDED FUNCTION-NAME PARAMETERS)
CHOICE        ::= (choice LABEL FOLDED MIN MAX SET SELECTED)
LABEL         ::= STRING
FOLDED        ::= BOOLEAN
FUNCTION-NAME ::= SYMBOL
PARAMETERS    ::= (EXPR*)
MIN           ::= NAT
MAX           ::= NAT
SET           ::= (EXPR*)
SELECTED      ::= (LINK*)
EXPR          ::= ACTION | S-EXPRESSION
```

The menus are trees built using `APPLY` and `CHOICE` nodes, and they are evaluated as usual arithmetic expressions, provided that all the nodes in the tree are *reducible*:

- An `ACTION` node is reducible if and only if its `ACTION` slot contains a reducible expression.
- An `APPLY` node is reducible if and only if all the `EXPR` objects contained in its `PARAMETERS` slot are reducible.
- A `CHOICE` node is reducible if and only if, n being the number of objects from the `SET` slot that have been interactively selected by the user, n belongs to the interval {`MIN`, `MAX`}, and if each one of these selected objects is also reducible.
- Any other kind of s-expression is always considered as reducible.

The evaluation of a given `MENU` node is enabled or disabled according to the value of its `EVALUATE` slot. The evaluation proceeds in the following way:

- A reducible `APPLY` node evaluates to the result of the call of the function related to the symbol contained in its `FUNCTION-NAME` slot, using the list contained in the `PARAMETERS` slot as the parameter list. This function can either be local or remote. The evaluation of an `APPLY` node can also include *side effects*, namely, *patches* to the current document. These side effects are stored in order to be performed only if the whole execution process succeeds.
- Each time the user selects or deselects an item in the `SET` slot of a given `CHOICE` node, its `SELECTED` slot is updated accordingly (i.e., a link to the newly selected or deselected part of the item is either added in or removed from the `SELECTED` slot, respectively). A reducible `CHOICE` node evaluates to its `SELECTED` slot.
- A reducible `MENU` node evaluates to the evaluation of its `ACTION` slot. If the evaluation is successful, the stored side-effects are triggered.

5.3 Function Calls

When a reducible `APPLY` node is to be evaluated, the evaluator first finds the name of the procedure in the list of procedure names of all the plug-ins [3]. Once the relevant plug-in has been found, the evaluator performs the remote call. Local calls are considered as a particular kind of remote calls and are therefore processed the same way. The evaluation of such a remote call proceeds in two steps:

- The current function call with all its parameters evaluated is directly sent (possibly across a socket) to the relevant plug-in. To do this, we use the `REMOTE-CALL` node defined below.
- The function call is then remotely evaluated, and the remote process sends back the result to TEXMACS. This result contains an s-expression which is the result itself, and the (possibly void) set of *side effects*, which is stored until the whole evaluation process succeeds.

  The needed elements for the function calls are

```
RESULT        ::= BOOLEAN | (error ...) | S-EXPRESSION
PATCH         ::= (patch LINK KEYED-ELEMENT)
SIDE-EFFECT   ::= PATCH
REMOTE-CALL   ::= (eval-apply @APPLY FUNCTION-NAME PARAMETERS)
REMOTE-RESULT ::= (result RESULT (SIDE-EFFECT*))
```

where `@APPLY` describes a slot which must contain a link to a fragment of the type `APPLY`.

6 Example

Let us now describe the behaviour of our interface using our running example. We suppose that the user wants to start with a proof of the deMorgan theorem:

```
(context (theory-by-name set-theory)
  { key=THEO1 }
  (theorem deMorgan
     (conclusion  _  <<∀A : set . ∀B : set . $\overline{A \cup B} = \overline{A} \cap \overline{B}$>>)))
```

First the user inserts the following markup containing a void proof for the theorem shown (using a button or a keyboard shortcut):

```
(with prog-language ‘‘core’’ prog-session ‘‘default’’
  { key=PROOF1 } (proof @THEO1))
```

When this markup is inserted into the document, the plug-in symbolically described in the attribute `prog-language` — here Core — is called and sends its answer in form of a patch back to TEXMACS. In our example, the new proof for the theorem plus the relevant environment (cf. Section 4.4) is

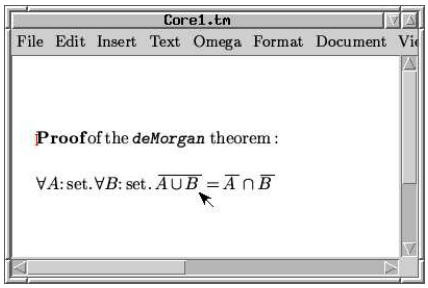
[3] The list of available procedures for each plug-in is sent to TEXMACS at *boot* time, that is, when the plug-in is used for the first time.

uploaded into Core which returns a patch containing the first proof step with its set of available assumptions. The markup of the patched document is then as follows [4]:

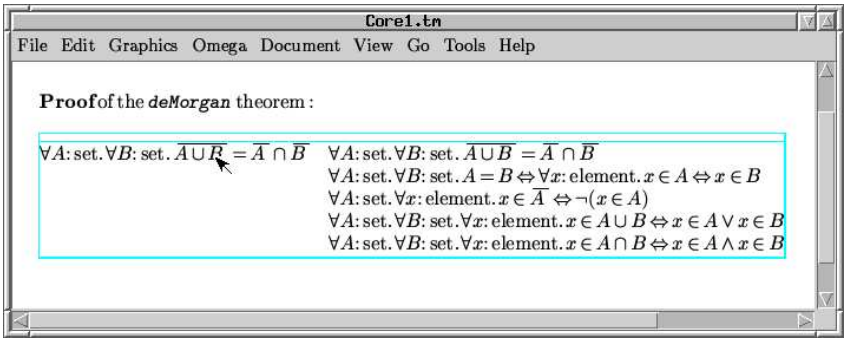
```
[...]
{ key=THEO1 }
(theorem deMorgan
  (conclusion _ { key=GOAL1 }<<∀A : set . ∀B : set . A ∪ B = A ∩ B>>))
[...]
(with prog-language ‘‘core’’ prog-session ‘‘default’’
  { key=PROOF1 }
  (proof @THEO1 (proof-step
    (@GOAL1
     ({key=HYP1}
      <<∀A : set . ∀B : set . A = B ⇔ ∀x ∈ E, x ∈ A ⇔ x ∈ B>>
      {key=HYP2}
      <<∀A : set . ∀x : element . x ∈ A ⇔ ¬(x ∈ A)>>
      {key=HYP3}
      <<∀A : set . ∀B : set . ∀x : element . x ∈ A ∪ B ⇔ x ∈ A ∨ x ∈ B>>
      {key=HYP4}
      <<∀A : set . ∀B : set . ∀x : element . x ∈ A ∩ B ⇔ x ∈ A ∧ x ∈ B>>)
     ())
    open ()
    { key=MENU1 }
    (menu false
      { key=APPLY1 }
      (apply _ true
             apply-rule
             ({ key=CHOICE1 }
              (choice _ false 2 2 (@GOAL1 @HYP1 @HYP2 @HYP3 @HYP4)
                ()))))))))
```

Note that in the `THEOREM` object, a key (cf. Section 4.3) has been added to the goal formula. Similarly, the available assumptions in the `PROOF-STEP` node have keys, which are used to refer to these formulae in the menu elements. Given the appropriate set of macros, our `PROOF` object with its menu is displayed in the TEXMACS editor as follows:

[4] Throughout this paper, the modified parts are shown as italics underlined text. In this first step, though, most of the markup is new, so we underline only the isolated new part for adding a key to the goal in the theorem, and the beginning of the `proof-step` node.



When the user clicks on the node representing the proof step, the root element in its menu tree (namely, the `APPLY1` node with its `CHOICE1` sub-node containing the goal followed by the set of available assumptions) unfolds itself as a menu on the right hand side of the proof step [5] :



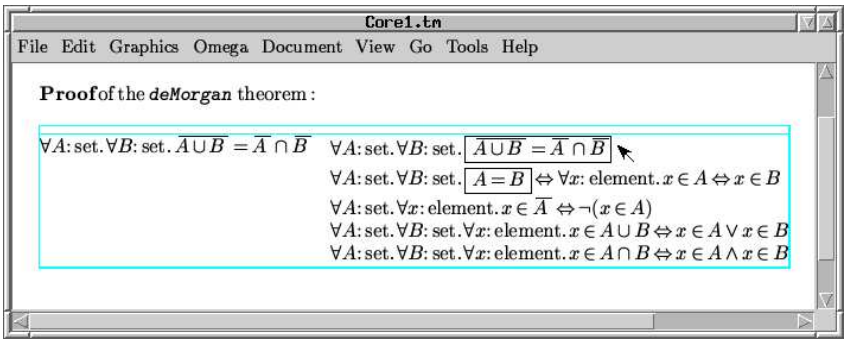
When the menu is unfolded, the related markup becomes:

```
{ key=MENU1 }
(menu false
  { key=APPLY1 }
  (apply _ false
         apply-rule
         ({ key=CHOICE1 }
          (choice _ false 2 2 (@GOAL1 @HYP1 @HYP2 @HYP3 @HYP4) ())))
```

Performing the Selections

When the user selects the appropriate sub-formulae among the set of available assumptions in the menu as indicated by boxes

[5] Note that the selections are only performed inside the menu on the right hand side of the proof step (i.e., no selection occurs in the node representing the proof step). Moreover, the selection mechanism allows for complex goal formulae, such that parts of the goal can be used to rewrite other parts of the goal. For instance, in a goal formula $x = 0 \Rightarrow x + y = y$ the sub-formula $x = 0$ can be applied to $x + y = y$.



the corresponding markup changes, namely (and if we suppose that the user starts first by performing her selections inside the chosen assumption, and next inside the goal) the SELECTED slot of the CHOICE1 choice first becomes:

```
{ key=CHOICE1 }
(choice _ false 2 2 (@GOAL1 @HYP1 @HYP2 @HYP3 @HYP4)
                    (@HYP1(2 2 1)))
```

When the user performs its selection inside the goal (i.e., the first element in the menu), we similarly obtain:

```
{ key=CHOICE1 }
(choice _ false 2 2 (@GOAL1 @HYP1 @HYP2 @HYP3 @HYP4)
                    (@GOAL1(2 2) @HYP1(2 2 1)))
```

Note that the CHOICE1 node is now reducible, because we have made two selections, namely @GOAL1(2 2) and @HYP1(2 2 1), as it is specified by the MIN/MAX-slots of CHOICE1. The APPLY1 and MENU1 nodes thus become also reducible, and the EVALUATE slot of the MENU1 menu can now be raised by the user (using an input action such as a double-click). The MENU1 menu thus becomes:

```
{ key=MENU1 }
(menu true
  { key=APPLY1 }
  (apply _ false
         apply-rule
         ({ key=CHOICE1 }
          (choice _ false 2 2 (@GOAL1 @HYP1 @HYP2 @HYP3 @HYP4)
                              (@GOAL1(2 2) @HYP1(2 2 1))))))
```

Evaluation of the Menu

The menu can now be evaluated. The CHOICE1 node is first reduced, and the following REMOTE-CALL node is created:

```
{ key=REMOTE1 }
(eval-apply
```

```
   @APPLY1 apply-rule
           ((@GOAL1(2 2) @HYP1(2 2 1))))
```

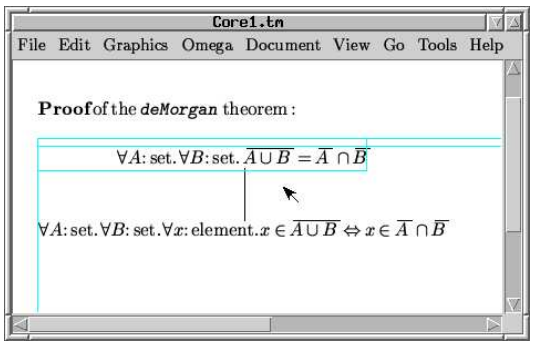
The `REMOTE1` node above is then sent to CORE, which performs the operation, and sends back the following `REMOTE-RESULT` expression as an answer to the TEXMACS side:

```
(result true
  ((patch @PROOF1(2 2)
     (apply <<A = B ⇒ ∀x : element . x ∈ A ⇔ x ∈ B>>))
   (patch @PROOF1(2 3)
     ((proof-step
        ({ key=GOAL2 }
          <<∀A : set . ∀B : set . ∀x : element .x ∈ A ∪ B ⇔ x ∈ A ∩ B>>
          (...) (...))
         open ()
         { key=MENU2 }(menu ...)))))))
```

This answer is a successful one, which is indicated in the first argument of the `(result ...)` expression by `true`. Since the `APPLY1` node, which corresponds to the successful `REMOTE1` remote call, is the root node of the `MENU1` menu, the patches can then be applied. The proof object `PROOF1` becomes:

```
{ key=PROOF1 }
(proof @THEO1
  (proof-step
  (@GOAL1 ( { key=HYP1 } <<...>> ...) ())
  (apply <<A = B ⇒ ∀x : element . x ∈ A ⇔ x ∈ B>>)
  ((proof-step
     ({ key=GOAL2 }
       <<∀A : set . ∀B : set . ∀x : element . x ∈ A ∪ B ⇔ x ∈ A ∩ B>>
       (...) (...))
     open ()
     { key=MENU2 } (menu ...)))
  { key=MENU1 } (menu ...)))
```

The physical markup is then updated, and the modified proof finally appears in the TEXMACS editor as follows:



7 Related Work

Our project is closely related to the TmCoq [6] project which aims at the full integration of TEXMACS with the theorem prover Coq. This project finds it highly desirable "to achieve the functionality allowing the user to interact with the prover in a mode as transparent as possible" [1] and that the "syntax conversions between the generic publication oriented editing tool TEXMACS and the theorem prover should be automated" [1]. We agree with both objectives, but remark that the former is probably of interest for expert Coq users only. For purposes of system-independent tutoring of mathematical proofs, or supporting non-expert users, a distinguished goal of our project is to have additionally a generic interface featuring a knowledge representation that allows one to connect various supporting tools. Both TmCoq and our project have many goals in common leaving room for fruitful collaboration.

Two important related interfaces to theorem provers are Proof General [7] and Pcoq [8]. The research in both projects has led to significant contributions such as proof-by-pointing, proof script management, and a distributed approach to user interfaces for theorem provers. Pcoq's main characteristics are a graphical interface, the possibility to run Pcoq and Coq as separate processes, structural editing of formulae and commands in combination with mouse-based navigation. Proof General is a generic interface for proof assistants, currently based on the customisable text editor Emacs. It shares many of the previously mentioned characteristics and it has been employed as interface to several mathematical assistance systems such as Isabelle [12], Coq [4], and LEGO [14].

The most closely related work with respect to the menu language is the generic proof editor Jape [16]. It provides a means to define a general menu to invoke the tactics and inference rules as well as to describe via pattern matching which tactic or inference rule to invoke if the user double-clicks on hypotheses or conclusions. The tactic language itself can be used to write simple search behaviours interleaved with querying whether the user has selected some text part and possibly include the marked text fragment as argument to tactics. In Jape tactics invoked by double-clicking can only use text-parts that have been selected by the user before the double-click. Hence, it seems that at most one user-selected text part can be used by the tactic. The nested menu structure presented in this article allows us to provide arbitrary many parameters to a tactic. Moreover, the set of available alternatives to choose can depend on other selected parameters and be computed on the fly by the prover. Furthermore, the menu language presented in this paper is deliberately not biased to any prover-specific functionality: it can be used to implement any interaction between TEXMACS and a background system, not

[6] http://tmcoq.audebaud.org/
[7] http://proofgeneral.inf.ed.ac.uk/
[8] http://www-sop.inria.fr/lemme/pcoq/

only for developing proofs. The reason is that we want to use the same menu structure also to query a knowledge base to find relevant theorems, or to get a list mathematical theories which use a specific logical symbol. Note that our interface allows the integration of the behaviour implemented in Jape as one *possible* means of interaction from the editor to the prover. Indeed, pattern matching of available rules or tactics is possible in our case by providing such a function in the prover which takes as argument some selected text. The results would be sent to TEXMACS as a patch to the actual proof.

8 Conclusion

In this article we reported about the ongoing integration of the scientific text editor TEXMACS with the proof assistant system ΩMEGA. The goal of the integration is to use ΩMEGA as context-sensitive reasoning and verification service accessible from within TEXMACS, where the proof assistant adapts to the style, in which an author would like to write its mathematical publication, and to hide any irrelevant system peculiarities from the user.

The kind of interface architecture described in this paper, built on a symbolic protocol layer and on a sophisticated display engine, is not completely unforeseen in the UITP community [5]. Nevertheless, to the best of our knowledge, such interfaces are still not widely used in the domain of theorem proving. This is why we build our system as a plug-in to TEXMACS [7], which on the one hand is a first class quality mathematical text editor, and on the other hand has also shell-style interfaces to many CASs but none for theorem provers (although some attempts in this direction have already been made [1]).

Based on a formal representation language, we presented an extended TEXMACS document format which allows for professional typesetting while the semantic content of the document is accessible for formal systems. Based on that static format for semantic mathematical documents, we developed a uniform language for the specification of context-sensitive menus, an evaluation semantics for menus to incrementally compute parameters for the actions specified in menus, and a diff/patch mechanism to propagate updates computed by the actions to the document.

Although currently the proofs are tailored to the rules of the Core calculus, the representation language in principle is parameterised over a specific language for proofs. We plan to replace the Core-specific proof languages by some generic notion of proofs, in order to obtain a generic format for formalised mathematical documents. To this end, we will start from a language for assertion-level proofs with underspecification [3], which we developed from previous experiences with tutorial dialogues about mathematical proofs between a computer and students [13].

We also envision the combination of our approach with natural language processing tools as recently investigated and developed, for instance, in projects such as Dialog [13]. Furthermore, it should be possible to integrate the sys-

tem with mathematics e-learning environments (e.g., ActiveMath [11]) and to connect to distributed mathematical knowledge repositories in the web (e.g., MBase [10]).

References

[1] Philippe Audebaud and Laurence Rideau. TEX$_{MACS}$ as authoring tool for formal developments. In Christoph Lüth and David Aspinall, editors, *Proceedings of the Workshop User Interfaces for Theorem Provers (UITP'03)*, Rome, Italy, September 2003.

[2] Serge Autexier. *Hierarchical Contextual Reasoning.* PhD thesis, Saarland University, Saarbrücken, Germany, 2003.

[3] Serge Autexier, Christoph Benzmüller, Armin Fiedler, Helmut Horacek, and Quoc Bao Vo. Assertion level proof representation with under-specification. *Electronic Notes in Theoretical Computer Science*, 93:5–23, 2004. Proceedings of the Mathematical Knowledge Management Symposium.

[4] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. The Coq proof assistant — reference manual version 6.1. Technical Report 0203, INRIA, May 1997.

[5] Yves Bertot. The CtCoq system: Design and architecture. *Formal Aspects of Computing*, 11(3):225–243, 1999.

[6] Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by pointing. In Masami Hagiya and John C. Mitchell, editors, *Symposium on Theoretical Aspects Computer Software (STACS)*, number 789 in Lecture Notes in Computer Science. Springer, 1994.

[7] Joris Van der Hoeven. Gnu TEXMACS: A free, structured, WYSIWYG and technical text editor. Number 39-40 in Cahiers GUTenberg, May 2001.

[8] Armin Fiedler. *User-adaptive Proof Explanation.* Phd thesis, Saarland University, Saarbrücken, Germany, 2001.

[9] Xiaorong Huang. *Human Oriented Proof Presentation: A Reconstructive Approach.* Number 112 in DISKI. Infix, Sankt Augustin, Germany, 1996.

[10] Michael Kohlhase and Andreas Franke. MBase: Representing knowledge and context for the integration of mathematical software systems. *Journal of Symbolic Computation*, 23(4):365–402, 2001.

[11] Erica Melis and Jörg H. Siekmann. ActiveMath: An intelligent tutoring system for mathematics. In Leszek Rutkowski, Jörg Siekmann, Ryszard Tadeusiewicz, and Lotfi A. Zadeh, editors, *Artificial Intelligence and Soft Computing — ICAISC 2004: 7th International Conference*, number 3070 in Lecture Notes in Computer Science, pages 91–101, 2004.

[12] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[13] Manfred Pinkal, Jörg Siekmann, Christoph Benzmüller, and Ivana Kruijff-Korbayova. Dialog: Natural language-based interaction with a mathematics assistance system. Project proposal in the Collaborative Research Centre SFB 378 on Resource-adaptive Cognitive Processes, 2004.

[14] Robert Pollack. *The Theory of LEGO — A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1996.

[15] Jörg Siekmann, Christoph Benzmüller, Armin Fiedler, Andreas Meier, Immanuel Normann, and Martin Pollet. *Proof Development in OMEGA: The Irrationality of Square Root of 2*, pages 271–314. Kluwer Applied Logic series (28). Kluwer Academic Publishers, 2003. ISBN 1-4020-1656-5.

[16] B. Sufrin and R. Bornat. User Interfaces for Generic Proof Assistants – Part I: Interpreting Gestures. In *In User interfaces for Theorem Provers*, York, UK, 1996.

[17] Joris van der Hoeven *et al.* The TeXmacs manual (stylesheet language). *http://www.texmacs.org/tmweb/manual/web-manual.en.html*, 1999-2005.