Certified and Portable Mathematical Documents from Formal Contexts

Olga Caprotti[1], Herman Geuvers[2], and Martijn Oostdijk[2]

[1] RISC-Linz Research Institute for Symbolic Computation, Johannes Kepler University, A-4020 Linz, Austria, olga.caprotti@risc.uni-linz.ac.at
[2] Computing Science Institute, University of Nijmegen, P.O.Box 9010, 6500GL, Nijmegen, The Netherlands, {herman,martijno}@cs.kun.nl

Abstract. This paper deals with the problem of generating interactive natural language documents based on formal mathematics. It describes how formal mathematical developments, carried out in the type theoretical theorem prover Coq, can be transformed to readable and interactive documents viewable using standard Web browser technology. The transformation process produces documents encoded in an XML application called OMDOC, suited for describing mathematical documents.

1 Introduction

Since the advent of the World Wide Web, interactive documents have become increasingly popular. Users expect content to be presented interactively, allowing non-linear browsing of the document and embedding of non-textual objects, enhancing exploration at multiple levels of detail. From the authoring point of view, it is yet unclear which is the best approach for preparing such documents. It seems as though the author has to take into account every possible future behavior of the reader and manually craft the interactivity. Clearly this means a considerable overhead in time. How can one create interactive documents without doing an exponential amount of extra work?

One solution might be to convert existing non-interactive documents into interactive ones by exploiting the inherent structure present in the document. For example, the document you are currently reading is not interactive. Yet, it also is not flat text, the document is structured in sections and within each section are references to the other sections. In an interactive version of this document these references would be presented as hyper-links, enabling the reader to jump to the section immediately. This form of interaction, called *internal interaction*, is based on the internal structure of the document. It requires that the author makes this structure explicit in the document: Sections have to be marked as sections, and references to sections as references. Fortunately, adding this kind of structure is already common practice when documents are prepared using a typesetting system like LaTeX. In general all the author has to do, is to be careful not to mix content and presentation, e.g. use `\section{Introduction}` rather than `\textbf{1 Introduction\\}`. If content and presentation are separated, an interactive version of the document is readily produced by appropriate conversion tools.

Now consider, instead of conventional textual documents, formal mathematical documents like those in a library of a theorem prover assistant. Because they inherently contain much more explicit structure, there are many more opportunities for interactivity. For example, if we have a definition $a := T$ in our document, then any future use of the definiens a can be seen as a *link* to the definiendum T. This piece of internal interaction does not have to be added afterwards, but is already present in the mathematical content. Proofs form another example where internal interaction may be derived from the mathematical content. If in a proof we apply another lemma, then this can be seen as a *link* to the statement (plus proof) of that lemma. Moreover, proofs have a lot of structure themselves (subcases, reasoning under a binder, hypothetical reasoning, etc.), which is left implicit in a textual presentation. The structure of proofs gives a way of *hiding* levels of detail of a proof (to be inspected on demand), thus providing a mechanism for *folding* and *unfolding* subproofs in the actual presentation. It is the formal mathematical content that dictates this folding and unfolding structure.

This paper deals with the creation of interactive mathematical documents based on such formal mathematical content. For the purposes of this paper we use formal mathematics as produced by the Coq theorem prover [3]. The type theoretic paradigm which forms the basis of Coq makes it a good candidate, as proofs are treated as first class citizens. However, the ideas also apply to other theorem provers since it is the mathematical structure which is exploited to create the interaction.

So far, we have mentioned *internal interaction*, a form of interaction based on the internal structure of the document. A different form of interaction, *external interaction*, allows the reader to send formal objects present in the document to back engines external to the documents. In our case we are interested in mathematical back engines such as theorem provers and computer algebra systems. The document acts as a user interface to such a back engine. The technology supporting communication to the back engines is readily available. Combining internal and external interaction does not pose any problems. In fact, both make similar demands on the author: the document has to be structured. Examples of external interaction can be found in *Algebra Interactive* [9], an interactive book on Algebra inviting students to try to solve example problems using a computer algebra system.

The OpenMath [18] language is designed to allow universal exchange of mathematical objects. It therefore facilitates external interaction very well by abstracting from the details of the back engine. For internal interaction based on formal mathematical developments, an extension of OpenMath, called OMDOC [14], is used. In OMDOC entire mathematical documents are encoded instead of mere objects. OMDOC is an XML application, which means that a standard Web browser (in combination with an appropriate stylesheet) may be used to view documents expressed in OMDOC. How this is done is described in more detail in Section 4.

In order to provide a non-trivial presentation of formal proofs, they are *verbalized* as natural language text. In the automated theorem prover community, it is quite common to attempt to *explain* automatically generated proofs. See for example the PROVERB system [13], and the Theorema system [5]. The natural language generation mechanism employed in Section 5 is based on algorithms described in [11, 10], which directly link fragments of natural language text to λ-terms. Even though the mechanism is crude, it still yields readable proofs, especially in combination with interactive adaptation of the level of detail. Improvements to make the verbalization more sophisticated are work in progress.

The remainder of this paper is organized as follows. Section 2 describes some elementary type theory and theorem proving based on type theory. Section 3 describes how an informal mathematical theory is turned into a formal context. Section 4 introduces XML, OpenMath and OMDOC. Section 5 provides details on how we generate the actual OMDOC documents. Finally, Section 6 gives the results and the conclusions.

2 Type Theory and Formal Mathematics

This section gives a short introduction to theorem proving based on type theory. The general basic ideas that make type theoretical theorem proving interesting for our purposes are recalled. The exposition is concerned with a fragment of the type theory used in the Coq system.

The logical calculus of Coq is called the *Calculus of Inductive Constructions* (CIC) [12], an extension of the *Calculus of Constructions* described for example in [2]. The fragment we use is almost all CIC, but for example no co-induction.

The key idea behind directly encoding mathematics in type theory is the 'propositions-as-types' or 'proofs-as-terms' correspondence, originally due to Curry, Howard and De Bruijn. This interpretation encodes proofs as typed λ-terms. Under this interpretation a statement '$M : A$', which is pronounced as 'M has type A' or 'A is inhabited by M', can be read in two ways, depending on the type of A:

- M is an *element of the set* denoted by A,
- M is a *proof of the proposition* denoted by A.

This duality has the advantage that checking the validity of a proof becomes as easy as checking the type of a term. Moreover, since the type checking algorithm can be very small, the risk of implementation errors is kept to a minimum and, as a result, the calculus is very safe. However, from the perspective of the current paper, this duality also has the drawback that the same primitive notions represent different concepts. Therefore, a type inference algorithm is needed to disambiguate the presentation of terms. Implementing a type inference algorithm for CIC is a relatively easy task.

Terms on their own are not enough to represent mathematics. Mathematics consists not only of expressions but of expressions occurring in a *context* relative to other expressions. A context is a list of *context items*. Each context item is either an *assumption* or a *definition*. An assumption is used to declare a new symbol with its type. It states a name for the new symbol and the type. The name can represent a mathematical object whose existence is assumed or it can represent an axiom which is assumed to hold. A definition is used to declare a new symbol with its type, but a definition also states a defining term for the symbol. The new symbol declared by an assumption or a definition may be used in the other context items in the remainder of the context. Contexts are generated by the following grammar.

Definition 1 *Abstract syntax for pseudo contexts. Let* $\mathbf{S}$ *be a countable set of individuals representing identifiers.*

$$\Gamma ::= \varepsilon \mid \Gamma, \mathbf{S} : \mathbf{T} \mid \Gamma, \mathbf{S} := \mathbf{T} : \mathbf{T}$$

Not every pseudo context generated by the above grammar is valid. There are a number of restrictions, mostly dealing with variables. A context is not valid if variables are declared more than once in the context or if they are used before they have been declared. We will not make these restrictions precise here.

Calculi like CIC are usually given in two stages. First, an easy context-free abstract syntax describing a large set of pseudo elements. Second, restrictions on the set of pseudo expressions, in the form of a type assignment system, to allow only those pseudo expressions representing meaningful mathematical terms. The type system also categorizes the typeable elements, gives them a partial meaning. We do not give a type system for CIC, instead we present the terms of CIC and describe their use in formalizing mathematical notions. See [12] for the rules of the type system of CIC.

Definition 2 *Abstract syntax for pseudo expressions. Let* $\mathbf{V}$ *be a countable set of individuals representing variables, and* $\mathbf{S}$ *the individuals representing identifiers defined or assumed in the context.*

$$\begin{aligned}\mathbf{T} ::= {} & \mathbf{V} \mid \mathbf{S} \mid \mathsf{Set} \mid \mathsf{Prop} \mid \mathsf{Type} \mid \mathbf{T}\mathbf{T} \mid \lambda \mathbf{V} : \mathbf{T}.\mathbf{T} \mid \Pi \mathbf{V} : \mathbf{T}.\mathbf{T} \mid \\ & \langle \mathbf{T} \rangle \, \mathsf{cases} \, \mathbf{T} \,.\{(\mathbf{T} \Rightarrow \mathbf{T})^*\} \mid \\ & \mathsf{fix} \, \mathbf{V} \, [\mathbf{V} : \mathbf{T}]^* : \mathbf{T} \,.\mathbf{T} \mid \\ & \mathsf{ind} \, \mathbf{V} \, [\mathbf{V} : \mathbf{T}]^* : \mathbf{T} \,.\{(\mathbf{V} : \mathbf{T})^*\}\end{aligned}$$

We do not provide formal semantics for these expressions, just some informal descriptions which are illustrated by examples below. The calculus contains the usual expressions found in typed λ-calculi: variables, symbols defined in the context, sorts Set, Prop, and Type, described below, function application, λ-abstraction to form functions, and Π-abstraction to form function types. Function application is a binary operator, but by using *currying*, function of higher arity can be emulated. λ-abstraction forms functions by abstracting a variable over a term. Π-abstraction forms dependent product types, which can be seen as a function type when it is not dependent, in which case it will be denoted using the $\rightarrow$ symbol.

What makes CIC the calculus of *inductive* constructions are cases, fix, and ind. These primitives deal with inductive types and recursive functions. Inductive types are introduced using the ind primitive. When using the COQ prover in practice, the ind construction can only be used in a definition. Such an inductive definition adds also the constructors of the type to the context.

The cases construction is used to eliminate values of inductive types by doing a case distinction. It takes a term and a number of pattern-value pairs. The value corresponding to the first pattern that matches the term is the outcome of the cases expression.

Using the combination of cases and fix, primitive recursive functions can be defined over inductive types. For every type defined using ind, COQ automatically adds an *elimination term* (or *recursor*) to the context. This term is defined in terms of cases and fix, see [12] for the details. The elimination term can be used to specify recursive functions over the inductive type. It is customary to use fix and cases explicitly when defining functions over an inductive datatype, whereas the elimination term is used in proving assertions over an inductive type. We will take this into account during verbalization, where the elimination terms are treated as primitive. Consider, as an example, the natural numbers defined inductively in CIC as follows:

Example 3 *The type of natural numbers* nat *is defined using an inductive type with constructors* O *and* S*. The* plus *function is defined as a fixpoint over* nat*.*

$$\begin{aligned}
\mathsf{nat} &:= \mathsf{ind}\, X : \mathsf{Set} \\
&\qquad\qquad \{\mathsf{O} : X,\ \mathsf{S} : X \rightarrow X\} \\
\mathsf{plus} &:= \mathsf{fix}\, f\, [n : \mathsf{nat}] : \mathsf{nat} \rightarrow \mathsf{nat}\,. \\
&\qquad\qquad \lambda m : \mathsf{nat}. \\
&\qquad\qquad\quad \langle \mathsf{nat} \rangle \mathsf{cases}\, n\,. \\
&\qquad\qquad\quad \left\{ \begin{array}{ll} \mathsf{O} & \Rightarrow m, \\ (\mathsf{S}\, p) & \Rightarrow (\mathsf{S}\,(f\, p\, m)) \end{array} \right\}
\end{aligned}$$

COQ puts an elimination term in the context for every inductive type. We do not specify how this term is generated in general, but describe it only for the natural numbers example above. The reader is referred to [12] for the general mechanism. It is important to note that the elimination term is not primitive but is defined (automatically) in terms of cases and fix. However, in Definition 13, when verbalizing proof-objects containing natelim, it is treated as if it were a primitive notion.

Example 4 *The elimination term for* nat *is generated automatically by* COQ *(defined in terms of* cases *and* fix*).*

$$\mathsf{natelim} := \ \lambda P : \mathsf{nat} \rightarrow \mathsf{Prop}.$$

$$
\begin{array}{l}
\lambda h_0 : (P\,\mathsf{O}).\lambda h_1 : (\Pi n : \mathsf{nat}.(P\,n) \rightarrow (P\,(\mathsf{S}\,n))).\\
\quad \mathsf{fix}\, f\, [n : \mathsf{nat}] : (P\,n).\\
\quad\quad \langle \lambda n : \mathsf{nat}.(P\,n)\rangle \mathsf{cases}\, n : \mathsf{nat}.\\
\quad\quad \left\{ \begin{array}{ll} \mathsf{O} & \Rightarrow h_0,\\ (\mathsf{S}\,m) & \Rightarrow (h_1\, m\, (f\, m)) \end{array} \right\}\\
: \ \Pi P : \mathsf{nat} \rightarrow \mathsf{Prop}.\\
\quad (P\,\mathsf{O}) \rightarrow (\Pi n : \mathsf{nat}.(P\,n) \rightarrow (P\,(\mathsf{S}\,n))) \rightarrow\\
\quad\quad \Pi n : \mathsf{nat}.(P\,n)
\end{array}
$$

The above inductive definition of the natural numbers introduces a new datatype nat of type Set into the context, together with the constructors O and S and the elimination term natelim. Similarly, in spirit of the 'propositions-as-types' correspondence, logical types can be encoded as inductive types.

Example 5 *Proposition and predicate logic in* CIC*.*

1. *As usual, universal quantification* $\forall x : A.B$ *is implemented as* $\Pi x : A.B$*, and implication* $A \rightarrow B$ *as* $\Pi x : A.B$ *(in the latter case* x *is a fresh dummy variable).*
2. *The other logical connectives are defined using inductive types with constructors representing the introduction terms.*

$$
\begin{array}{rl}
\mathsf{False} & := \mathsf{ind}\, X : \mathsf{Prop}\\
 & \quad \{\}\\
\mathsf{not} & := \lambda a : \mathsf{Prop}.a \rightarrow \mathsf{False}\\
\mathsf{and} & := \mathsf{ind}\, X\, [a, b : \mathsf{Prop}] : \mathsf{Prop}\\
 & \quad \{\mathsf{andintro} : a \rightarrow b \rightarrow (X\, a\, b)\}\\
\mathsf{or} & := \mathsf{ind}\, X\, [a, b : \mathsf{Prop}] : \mathsf{Prop}\\
 & \quad \{\mathsf{orintrol} : a \rightarrow (X\, a\, b),\ \mathsf{orintror} : b \rightarrow (X\, a\, b)\}\\
\mathsf{ex} & := \mathsf{ind}\, X\, [A : \mathsf{Set}; P : A \rightarrow \mathsf{Prop}] : \mathsf{Prop}\\
 & \quad \{\mathsf{exintro} : \Pi x : A.(P\, x) \rightarrow (X\, A\, P)\}
\end{array}
$$

3. *Elimination terms are generated automatically in terms of* fix *and* cases*. We only give the types, the terms are similar to* natelim *in Example 4.*

$$
\begin{array}{rl}
\mathsf{falseelim} : & \Pi P : \mathsf{Prop}.\mathsf{False} \rightarrow P\\
\mathsf{andelim} : & \Pi A, B, P : \mathsf{Prop}.(A \rightarrow B \rightarrow P) \rightarrow A \wedge B \rightarrow P\\
\mathsf{orelim} : & \Pi A, B, P : \mathsf{Prop}.(A \rightarrow P) \rightarrow (B \rightarrow P) \rightarrow A \vee B \rightarrow P\\
\mathsf{exelim} : & \Pi A : \mathsf{Set}; P : A \rightarrow \mathsf{Prop}; P_0 : \mathsf{Prop}.(\Pi x : A.(P\, x) \rightarrow P_0) \rightarrow\\
 & \quad (\mathsf{ex}\, A\, P) \rightarrow P_0
\end{array}
$$

The constructors of those inductive types behave like introduction predicates. The elimination terms are defined in terms of cases and fix.

As a final example of the expressive power of CIC, consider the general polymorphic equality of COQ. Again, the introduction and elimination terms are treated as primitives during verbalization of proof-objects in which they occur.

Example 6 *Inductive equality.*

1. *Inductive equality is defined using an inductive type with one constructor representing the reflexivity axiom.*

$$\mathsf{eq} := \mathsf{ind}\, X\, [A\colon \mathsf{Set}; x\colon A] : A \to \mathsf{Prop}$$
$$\{\mathsf{eqintro}\colon (\mathsf{eq}\, x\, x)\}$$

2. *The elimination term is generated automatically defined in terms of* fix *and* cases*. We only give the type, the term is similar to* natelim *in Example 4.*

$$\mathsf{eqelim}\colon\ \Pi A\colon \mathsf{Set}.\Pi x\colon A.\Pi P\colon A \to \mathsf{Prop}.$$
$$(P\, x) \to \Pi y\colon A.((\mathsf{eq}\, x\, y) \to (P\, y))$$

As stated above, the typing rules are not given here. These typing rules restrict the pseudo expressions to well-formed expressions: Expressions that do not make sense are simply not assigned any type. The rules induce a structure on the expressions, dividing them in three classes of typeable expressions: The

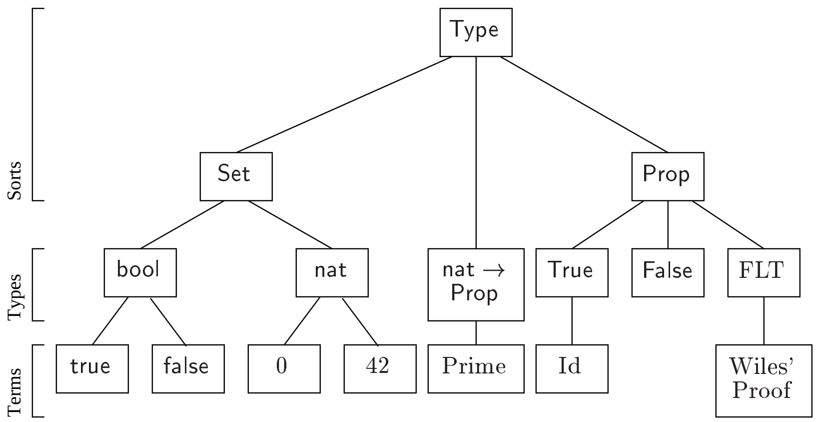

Fig. 1. Sorts, Types, and Terms.

sorts level, the *types* level, and the *terms* level. In Figure 1 some examples of sorts, types, and terms are displayed. Every expression in CIC falls in one of these three levels. Most objects defined in a typical formalization of a mathematical document live in the terms or types level.

CIC has three *sorts*: Set, Prop, and Type. Sorts serve as type for expressions on the types level. The sort Set is the type of datatypes such as nat. The sort

Prop is the type of propositions. The sort Type is the type of types but also serves as a type for the other two sorts.

Inhabitants of sorts (except for the sorts themselves) live on the types level. Examples of expressions living on the types level are the type of booleans bool and the type of natural numbers nat. Both these datatypes inhabit sort Set. An example of a type inhabiting the Type sort is nat $\rightarrow$ Prop which is the type of unary predicates over the natural numbers. Note that in CIC propositions are not identical to datatypes, they merely exist on the same level. It is easy to distinguish between them since propositions are characterized by the fact that they have type Prop.

Expressions which inhabit types are called terms and live on the lowest level. Terms are objects that do not have inhabitants themselves. Terms that inhabit a proposition (a type inhabiting Prop) are called proof-objects. A proposition can have many different inhabitants, representing different proofs of the proposition.

Note that these three levels are not syntactical classes, they are induced by the typing relation. Also, the interpretation of certain objects as datatypes, propositions, or proofs is arbitrary. Other interpretations of the sorts are possible, leading to different encodings of the same mathematical theories in CIC. However, the libraries of definitions and lemmas that come with COQ assume the interpretation sketched above, as do we when we verbalize proofs when generating interactive documents.

The typing rules from [12] are easily transformed into an algorithm for type-assignment. This is used in Section 5 to determine how to verbalize an object.

3 Generating Certified Contexts

Theorem provers like COQ are used in creating certified formal mathematical developments. Creating such a formal development involves encoding informal mathematical concepts, which requires the user to make many non-trivial implementation choices. Moreover, also the proofs have to be encoded in type theory, which is laborious.

Definitions of mathematical objects are encoded directly by the user. However, to assist the user in the proving process, many theorem provers offer a friendlier mechanism. In COQ a proof-object may be constructed in an interactive fashion. The user applies *tactics* to the current goal, which initially is the statement to prove. A tactic transforms a proof-object under construction (containing holes) into another proof-object under construction. Initially the proof-object is nothing but a hole. The types of the holes are the goals left to prove, so a tactic also transforms old goals into new ones. When finally no more holes are left in the proof-objects, it is checked to see whether it is really a proof of the statement.

The result of a COQ session is a formally verified context. Inside COQ this context is stored in some internal data structure, but for us it is not really necessary to know what the internal data structure is like. This context is parsed and stored in our tool as an instance of the *CoqTree* class described in [17]. Each

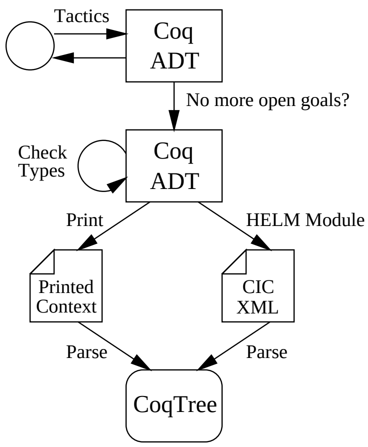

Fig. 2. Overview of the context generation process.

term occurring in the context is implemented as an object of this class. The *CoqTree* class has methods to do reduction, and type inference.

There are two ways to obtain a *CoqTree* object starting from the Coq ADT. One way is printing the context with Coq's `Print` command, and parsing the 'printed context' with an ad hoc parser. The other way is to use the Coq module made by the HELM project group which exports the Coq terms as an XML document [1]. To this end, they have developed a XML based language, extending the MathML language, for terms of the Calculus of Inductive Constructions. The resulting XML document is then used to generate a *CoqTree* object for the context. The latter method is obviously cleaner as the XML is generated directly from the Coq ADT, rather than through a context which is pretty-printed for human consumption.

Based on the resulting *CoqTree* class our implementation generates an OMDOC document, as described in Section 5. The next section introduces XML, OpenMath and OMDOC.

4 XML, OpenMath, and OMDOC

This section describes XML and two XML applications for specification of mathematical content, OpenMath and its extension OMDOC. Both the design goals and a large part of the syntax are described here. However, a complete description of these languages is outside the scope of this paper and can be found in the literature and online [18, 19, 14, 16, 21]. Here, we limit the exposition to the features we actually use.

The *extensible markup language*, XML [21], has emerged as a preferred language for storing and communicating structured content over the Internet. It is a meta language, in the sense that it describes other languages. XML grammars are called *document type definitions* or DTDs, and prescribe the elements appearing in documents written in the specific instance of XML. The XHTML language is an example of an XML application. When appropriate, an XML document written according to one DTD can be transformed into an XML document of different DTD using a stylesheet. Stylesheets are formulated in XSL, the XML stylesheet language, which happens to be an XML application itself. When transforming to XHTML, the stylesheet can be seen as a specification of how documents should be presented. This allows a rigorous distinction between presentation and content. The content is specified in the XML document according to a previously defined DTD, the presentation is specified in a stylesheet.

There are many advantages to using XML as a metalanguage. Open standards which exceed the proprietary formats of individual systems are useful and will become more and more important as we move toward an interconnected information society. XML makes designing such open standards a little bit easier. Parsing is potentially easier as technology from others can be used, which means parsing can be done on a higher level. Other tools such as viewers and editors can also be reused. For our purposes XML is ideal, as it allows us to draw the distinction between content and presentation explicitly in the document. It is not surprising the computer algebra and theorem proving communities embrace XML for encoding the OpenMath and OMDOC standards described below.

The OpenMath standard [18, 19] is a language intended to share expressions between computer mathematics systems. It provides primitives for variables, abstraction and application. In order to be universally applicable, OpenMath theories can be parameterized with a so-called *content dictionary* (CD) containing names and types for new symbols. A CD for type theory is available, see [6]. The OpenMath language, for the purposes of this paper, consists of the following elements.

`<OMV>`. Variables.
`<OMS>`. Symbols are defined in the CD.
`<OMA>`. Function application of symbols.
`<OMBIND>`. Binding of variables. Useful for λ- and Π-abstraction.
`<OMATTR>`, `<OMATP>`. Attributes, can be assigned to the other OpenMath elements. Useful for type information.

OpenMath is useful for communicating mathematical objects between computer algebra systems. The problem with OpenMath is that it only describes objects not theories or documents or contexts. The ability to encode contexts is essential if one wants to communicate with theorem prover systems. The CD mechanism of OpenMath is not flexible enough for theorem provers: Contexts in which one can define new concepts are needed. The solution lies in an extension of OpenMath called OMDOC.

The OMDOC language [14, 16] is an extension of OpenMath which is used to describe mathematical documents rather than mathematical terms. It uses OpenMath to express individual terms. An important property of OMDOC documents is that they may also contain informal textual parts. The OMDOC language, for the purposes of this paper, consists of the following elements:

<symbol> encloses a new mathematical symbol. In the remainder of the document the symbol may be used in OpenMath objects as if it was declared in a CD[1].

<definition> encloses properties of symbols declared earlier in the document. It is not required to give a definition in "dictionary style" where a symbol is assigned an expression which completely defines it.

<axiom> encloses a new axiom.

<assertion> encloses a new assertion which may be proved later in the document. The type of assertion can be refined to lemma or theorem using the `type` attribute.

<proof> encloses a proof of an assertion. An assertion can have any number of proofs. Proofs consist of derive steps and have a conclusion. Furthermore, proofs may contain recursive proofs.

<derive> encloses a derive step inside a proof.

<conclude> encloses the conclusion of a proof. Proofs end in a conclusion indicating what statement was proved.

<CMP> and <FMP> enclose commented and formal mathematical properties, respectively. These are used inside the other elements.

To summarize, the use of XML for marking up the semantic structure of a document supports a rigorous distinction between content and presentation. The XML applications OpenMath and OMDOC are good candidates for representing mathematical content as both the expressions and the meta-structure of a mathematical document can be dealt with.

5 Generating Portable Documents

This section describes how OMDOC documents are generated from COQ contexts. The goal is to generate certified and portable mathematical documents from COQ contexts. It should be clear, from what has been said above, that starting from formally developed contexts ensures the correctness of the mathematical content expressed in the document: the result is certified by a mathematical assistant. The usage of XML as the format in which the document is exported will take care of portability by abstracting the content from the syntactical notations of the specific system used in the initial stage of generation of the document.

It is our intention that the documents are readable as natural language mathematical documents, so some verbalization of the formal objects is needed. Generating convincing natural language text from formal mathematical representations is not our main interest. However, a straight-forward verbalization algorithm is available which already yields interesting documents. The informal natural language text is embedded in the resulting OMDOC document.

Essentially, this means that content and presentation in the document are stored and handled separately as suggested in [7]. Since OMDOC is XML, XSL

[1] The name of the CD is by default the name of the OMDOC theory within which the symbol's definition resides.

stylesheets can be used to produce the actual presentation. In Figure 3 an overview of the approach used to generate OMDOC documents is given. We explain the figure and describe its components in more detail in the rest of this Section. Boxes with rounded corners represent JAVA classes in our implementation. Once

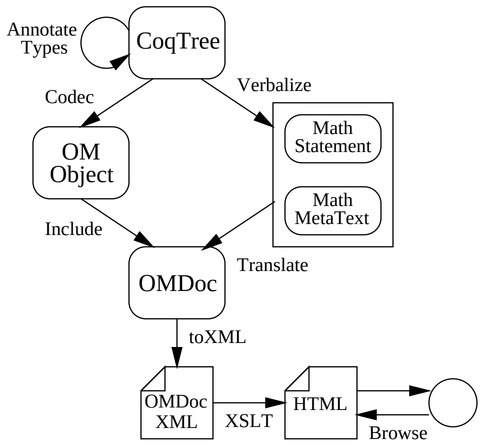

Fig. 3. Overview of the document generation process.

a *CoqTree* of the context is obtained, as described in Section 3, it is annotated with type information, more exactly in every subterm the type of that subterm is stored. In computing the type we take into account the type which is explicitly given in the context, i.e. we annotate the subterms with the *expected* type, as defined in [10]. The algorithm for verbalization of proofs relies on this type information.

A *MathStatement* consists of *MathUnit*s containing either informal text or formal *OMObject*s.

Notation 7 *MathStatement.*

1. *The construction of a MathStatement out of MathUnits* $u_1, \ldots, u_n$ *is denoted as:* $\langle u_1, \ldots, u_n \rangle$.
2. *Concatenation of MathStatements is denoted by the* $+$ *symbol.*

The embedding of text units is still a bit crude. It would be more flexible and sophisticated to choose the level of units more abstractly, for example units like punctuation, adjectives, nouns, verbs. This would make presentation more flexible, and allow more subtle natural language generation. It might be interesting to compare this to work such as [15], which attempts to provide a semi-formal level of description somewhere between natural language and type theory.

For proofs yet another intermediate datatype, named *MathMetaText*, is used. This datatype models the `<proof>` context item, and facilitates the internal interaction by which proofs can be inspected on multiple levels of detail. A *MathMetaText* consists of a list of *MathStatements* but may also contain recursive *MathMetaTexts*. Furthermore, every *MathMetaText* has a conclusion, which is just a *MathStatement*.

Notation 8 *The construction of a MathMetaText with conclusion s_0 out of MathStatements $s_1, \dots, s_n$ is denoted as:*

$$\begin{bmatrix} s_1 \\ \vdots \\ s_n \end{bmatrix}_{s_0}$$

The recursive nature of a *MathMetaText* provides the multiple levels of detail of a mathematical proof and also indicates the points in the representation where a proof can be folded or unfolded. If we look at the example *MathMetaText* in Notation 8, this object can either be *folded*, in which case we just display s_0 or *unfolded*, in which case we display $s_1, \dots, s_n$. If s_1 itself is again a *MathMetaText* $[t_1, \dots, t_m]_{t_0}$, then s_1 can also be folded (displaying t_0) or unfolded (displaying $t_1, \dots, t_m$). Whether specific objects are folded or unfolded is determined by the reader via the interface, but the author might provide a *preferred setting* along with the document, determining the statuses of all the folding/unfolding points at start-up.

To summarize, for the statement of an axiom or theorem (encoded by the type of a definition) we use *MathStatement*, and for the proof of a theorem we use *MathMetaText*. For non-propositional objects we just use *OMObjects* by employing the Coq-to-OpenMath Codec.

Here is how we verbalize propositional statements as a *MathStatement*. Propositional statements are generated from objects of type Prop. Note that this verbalization treats the propositional connectives as though they were primitives, even though they have a definition in Coq, see Example 5.

Definition 9 *Verbalization of a propositional statement as a MathStatement.*

$$\begin{aligned}
\mathsf{s}(\mathsf{not}\, A) &= \langle\, \text{“}\textit{not}\text{”} \rangle + \mathsf{s}(A) \\
\mathsf{s}(\mathsf{and}\, A\, B) &= \mathsf{s}(A) + \langle\, \text{“}\textit{and}\text{”} \rangle + \mathsf{s}(B) \\
\mathsf{s}(\mathsf{or}\, A\, B) &= \mathsf{s}(A) + \langle\, \text{“}\textit{or}\text{”} \rangle + \mathsf{s}(B) \\
\mathsf{s}(A \to B) &= \langle\, \text{“}\textit{if}\text{”} \rangle + \mathsf{s}(A) + \langle\, \text{“}\textit{then}\text{”} \rangle + \mathsf{s}(B) \\
\mathsf{s}(\forall x \colon A.B) &= \langle\, \text{“}\textit{for all}\text{”}, x, \text{“}\textit{in}\text{”}, A \rangle + \mathsf{s}(B) \\
\mathsf{s}(\mathsf{ex}\, A\, (\lambda x \colon A.B)) &= \langle\, \text{“}\textit{there exists an}\text{”}, x, \text{“}\textit{in}\text{”}, A, \text{“}\textit{such that}\text{”} \rangle + \mathsf{s}(B)
\end{aligned}$$

In the above verbalization rules all connectives are lifted to natural language. What remains are the atomic propositions, which are applications of predicates to mathematical objects. The atomic propositions are encoded as OpenMath objects. Alternatively the entire propositional statement could have been encoded as an OpenMath object. To which degree propositional statements belong to the object language or to the meta language seems to be a matter of taste. Perhaps this choice should be left to the reader of the document, this option is easily implemented as an internal interaction feature.

According to the rules stated in Definition 9, implication chains result in unnatural nested if-then-else verbalizations. To counter this, de-currying is applied. This means that a statement of the form $A \to (B \to C)$ is replaced by $(A \wedge B) \to C$ prior to verbalization. Something similar can be done to verbalize nested universal and existential quantifiers abstracting variables over the same domain. For example, the statement $\forall x\colon A.\forall y\colon A.B$ is verbalized as $\langle\text{"for all"}, x, \text{"and"}, y, \text{"in"}, A\rangle + \mathsf{s}(B)$.

Proof-objects are verbalized via the *MathMetaText* datatype. An adaptation of Coscoy's translation from [11, 10] is used. In the definition below, where appropriate, types of terms are indicated as superscripts.

Definition 10 *Verbalization of a proof-object as a MathMetaText. We use the verbalization of propositional statements* $\mathsf{s}(-)$ *as given in Definition 9. The type of the proof-object is assumed to be* τ *in each case.*

$$\mathsf{t}(h) = \Big[\langle\textit{"by"}, h, \textit{"we have"}\rangle + \mathsf{s}(\tau)\Big]_{\mathsf{s}(\tau)}$$

$$\mathsf{t}(M^{\forall x:A.B}N) = \begin{bmatrix}\mathsf{t}(M)\\ \langle\textit{"by taking"}, N, \textit{"for"}, x, \textit{"we get"}\rangle + \mathsf{s}(\tau)\end{bmatrix}_{\mathsf{s}(\tau)}$$

$$\mathsf{t}(M^{A\to B}N) = \begin{bmatrix}\mathsf{t}(N)\\ \mathsf{t}(M)\\ \langle\textit{"we deduce"}\rangle + \mathsf{s}(\tau)\end{bmatrix}_{\mathsf{s}(\tau)}$$

$$\mathsf{t}(\lambda h\colon A^{\mathsf{Prop}}.M) = \begin{bmatrix}\langle\textit{"assume"}\rangle + \mathsf{s}(A) + \langle\textit{"("}, h\,\textit{")"}\rangle\\ \mathsf{t}(M)\\ \langle\textit{"we have proved"}\rangle + \mathsf{s}(\tau)\end{bmatrix}_{\mathsf{s}(\tau)}$$

$$\mathsf{t}(\lambda x\colon A^{s}.M) = \begin{bmatrix}\langle\textit{"consider an arbitrary"}, x, \textit{"in"}, A\rangle\\ \mathsf{t}(M)\\ \langle\textit{"we have proved"}\rangle + \mathsf{s}(\tau)\end{bmatrix}_{\mathsf{s}(\tau)}$$

Proof-objects containing repeated λ-abstracted variables from the same domain can be presented in a more compact way by de-currying, similar to the decurrying used in verbalization of propositional statements containing repeated universal or existential quantifiers, see the remarks following Definition 9.

The next definition concerns verbalization of proof-objects involving cases.

Definition 11 *Verbalization of a proof-object as a MathMetaText,* cases*-case.*

$$\mathsf{t}\left(\begin{array}{l}\langle\tau\rangle\,\mathsf{cases}\,M\,.\\ \left\{\begin{array}{c}P_1 \Rightarrow B_1,\\ \vdots\\ P_n \Rightarrow B_n\end{array}\right\}\end{array}\right) = \begin{bmatrix}\langle\textit{"we reason by cases on"}, M\rangle\\ \langle\textit{"if"}, M, \textit{"is"}, P_1\rangle\\ \mathsf{t}(B_1)\\ \vdots\\ \langle\textit{"if"}, M, \textit{"is"}, P_n\rangle\\ \mathsf{t}(B_n)\\ \langle\textit{"in all cases"}\rangle + \mathsf{s}(\tau)\end{bmatrix}_{\mathsf{s}(\tau)}$$

There is no instance for the fix construction. The combination of fix and cases occurring in a proof-object suggests that the proof is by induction over an inductive type. At this moment we do not have a good solution for verbalization of such proofs. For some of the inductive types in the Coq standard library we have special verbalization rules. We can detect such situations as the generated elimination term (e.g. $\mathsf{natelim}$) will be used, see Definition 13. We also give special verbalization rules for introduction of the logical connectives.

Definition 12 *Verbalization of proof-objects involving logical connectives (using the constructors described in Examples 5,6).*

$$\mathsf{t}(\mathsf{andintro}\, A\, B\, a\, b) = \begin{bmatrix} \mathsf{t}(a) \\ \mathsf{t}(b) \\ \langle \textit{“therefore”} \rangle + \mathsf{s}(\tau) \end{bmatrix}_{\mathsf{s}(\tau)}$$

$$\mathsf{t}(\mathsf{orintrol}\, A\, B\, a) = \begin{bmatrix} \mathsf{t}(a) \\ \langle \textit{“therefore”} \rangle + \mathsf{s}(\tau) \end{bmatrix}_{\mathsf{s}(\tau)}$$

$$\mathsf{t}(\mathsf{orintror}\, A\, B\, b) = \begin{bmatrix} \mathsf{t}(b) \\ \langle \textit{“therefore”} \rangle + \mathsf{s}(\tau) \end{bmatrix}_{\mathsf{s}(\tau)}$$

$$\mathsf{t}(\mathsf{exintro}\, A\, P\, a\, H) = \begin{bmatrix} \mathsf{t}(H) \\ \langle \textit{“therefore”} \rangle + \mathsf{s}(\tau) \end{bmatrix}_{\mathsf{s}(\tau)}$$

$$\mathsf{t}(\mathsf{eqintro}\, A\, a) = \begin{bmatrix} \langle \textit{“by reflexivity”} \rangle + \mathsf{s}(\tau) \end{bmatrix}_{\mathsf{s}(\tau)}$$

Definition 13 *Verbalization of proof-objects eliminating connectives (using the elimination terms described in Examples 3,5,6).*

$$\mathsf{t}(\mathsf{falseelim}\, P\, f) = \begin{bmatrix} \mathsf{t}(f) \\ \langle \textit{“ex falso sequitur quodlibet”}, \\ \qquad\qquad \textit{“so we have”} \rangle + \mathsf{s}(\tau) \end{bmatrix}_{\mathsf{s}(\tau)}$$

$$\mathsf{t}(\mathsf{andelim}\, A\, B\, P\, f\, c) = \begin{bmatrix} \mathsf{t}(c) \\ \mathsf{t}(f) \\ \langle \textit{“so we have”} \rangle + \mathsf{s}(\tau) \end{bmatrix}_{\mathsf{s}(\tau)}$$

$$\mathsf{t}(\mathsf{orelim}\, A\, B\, P\, f_a\, f_b\, d) = \begin{bmatrix} \mathsf{t}(d) \\ \mathsf{t}(f_a) \\ \mathsf{t}(f_b) \\ \langle \textit{“in any case we have”} \rangle + \mathsf{s}(\tau) \end{bmatrix}_{\mathsf{s}(\tau)}$$

$$\mathsf{t}(\mathsf{exelim}\, A\, P\, P_0\, f\, x) = \begin{bmatrix} \mathsf{t}(f) \\ \mathsf{t}(x) \\ \langle \textit{“so we have”} \rangle + \mathsf{s}(\tau) \end{bmatrix}_{\mathsf{s}(\tau)}$$

$$\mathsf{t}(\mathsf{eqelim}\, A\, X\, P\, h\, Y\, q) = \begin{bmatrix} \mathsf{t}(h) \\ \mathsf{t}(q) \\ \langle \textit{“replacing”}, X, \textit{“with”}, Y, \\ \textit{“we get”} \rangle + \mathsf{s}(\tau) \end{bmatrix}_{\mathsf{s}(\tau)}$$

$$\mathsf{t}(\mathsf{natelim}\, P\, H_b\, H_s\, n) = \begin{bmatrix} \langle \textit{“induction on”}, P\rangle \\ \mathsf{t}(H_b) \\ \mathsf{t}(H_s) \\ \langle \textit{“so we have”}\rangle + \mathsf{s}(\tau) \end{bmatrix}_{\mathsf{s}(\tau)}$$

To facilitate the translation to OMDOC, we implemented JAVA classes corresponding to the different elements that can occur in an OMDOC document. These classes have a `toXML()` method which generates the OMDOC output. Each context item in the *CoqTree* context is translated to an OMDOC context item object. Here the type information present in the *CoqTree* is used to make the distinction between parameter assumptions and definitions on the one hand, and axioms and proofs of theorems on the other. In more detail: an assumption of propositional sort, $x : A : \mathsf{Prop}$, becomes an `<axiom>` element, and an assumption $x : A$ of non-propositional sort becomes a `<symbol>` element. A definition $x := M : A$ becomes an `<assertion type="theorem">` element with a `<proof>` element if A: Prop, and a `<symbol>` element with a `<definition>` element otherwise.

All these OMDOC context items have a formal `<FMP>` parts as well as a commented `<CMP>` part. The `<FMP>` part of definitions and assumptions is easy. We translate the *CoqTree* to an object of class *OMObject* using the COQ-to-OpenMath Codec described in [6].

The `<CMP>` part may contain formal OpenMath encoded terms but it may also contain flat text. The contents of the `<CMP>` parts is generated using the intermediate *MathStatement* datatype.

The *MathMetaText* objects representing proof-objects generate `<proof>` elements where each *MathStatement* corresponds to a `<proofstep>` element, and recursive *MathMetaTexts* become new `<proof>` elements. The *MathStatements* generate `<CMP>` parts. Formal units are replaced by OpenMath objects, textual units remain text. Therefore even though the `<CMP>` elements are commented content, they do contain formal parts.

Finally the OMDOC XML can be transformed to Dynamic HTML using an XSL stylesheet. Dynamic HTML supports folding and unfolding of trees which is used in interactively displaying of proofs.

6 Results and Conclusions

We have developed a method to create interactive mathematical documents based on formal contexts prepared in the COQ theorem prover. The formal structure of the mathematics is still present in the resulting document, but it can be presented as informal natural language text to the end user. The presence of formal structure in the document allows internal interaction, a form of interaction based on structured content. For example, using the fact that proof-objects are treated as first class citizens in the type theory of COQ, our implementation allows the reader to interact with the proofs in the document by adjusting the level of detail. For the proof-objects we first generate an intermediate level of *MathMetaText* objects. The natural language generation mechanism is based on

Coscoy's translation [11, 10] of proof-objects. This translation is a direct transduction algorithm, which often leads to artificially sounding natural language texts. However, the intermediate level allows the reader to adjust the level of detail in the presented proof text, and thus implements a form of internal interaction. The intermediate level offers opportunities for refinement making the translation more subtle.

In the implementation many emerging technologies are used. Java is used as the implementation language for most of the components, XML applications are used to communicate content between components. The resulting document is specified in an XML application called OMDOC. The use of Java and XML makes sharing code with other authors easy. The COQ-to-OpenMath encoder in [6] is used for displaying the formal mathematical terms occurring in a context. The encoder in turn makes use of the OpenMath library from the PolyMath group [20] to render OpenMath objects as XML. The OMDOC Java package we implemented follows closely the structure of the OMDOC DTDs, it consists of about 20 classes describing all the OMDOC elements we use. In generating the final interactive documents, we use the XSLT transformation program from Apache and XML APIs from IBM and Sun. The stylesheets are developed for OMDOC by the Omega project group. In the near future we may start using the HELM module, should also be easy and makes us less dependent on COQ.

We applied this technology to some small examples found in [8]. In future work we plan to apply it to bigger examples. More future work can be found in making the intermediate level of *MathStatement*s more subtle, maybe draw some inspiration from Nederpelt's Pseudo Type Theory [15] or the proof-planning provers such as Omega [4]. Both of these seem to provide a level between formal logic based mathematics and informal natural language, although they are more interested in the opposite direction, i.e. going from informal to formal. For longer term future work we also would like to investigate the translation from (a subset of) OMDOC to COQ, but presentation of formal mathematics already offers many challenges.

References

1. A. Asperti, L. Padovani, C. Sacerdoti Coen, and I. Schena. Towards a Library of Formal Mathematics, August 2000. Poster session at TPHOLs 2000.
2. H. P. Barendregt. Lambda calculi with types. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 2, pages 117–309. Oxford University Press, 1992.
3. B. Barras et al. *The Coq Proof Assistant, Reference Manual, Version 6.3.* INRIA-Rocquencourt - CNRS-ENS Lyon, July 1999.
4. C. Benzmüller et al. Omega: Towards a mathematical assistant. In W. McCune, editor, *Proceedings CADE-14*, volume 1249 of *LNAI*, pages 252–255, Townsville, Australia, 1997. Springer-Verlag.
5. B. Buchberger et al. A survey on the theorema project. In *Proceedings of ISSAC'97*. ACM Press, New York, 1997.

6. O. Caprotti and A. M. Cohen. Integrating computational and deduction systems using OpenMath. In *Proceedings Calculemus'99*, volume 23 of *ENTCS*, Trento, Italy, July 1999. Elsevier.
7. O. Caprotti and M. Oostdijk. On communicating proofs in interactive mathematical documents. In J. A. Campbell and E. Roanes-Lozano, editors, *Proceedings of AISC 2000*, volume 1930 of *LNAI*, pages 53–64. Springer-Verlag, July 2000.
8. O. Caprotti and M. Oostdijk. Formal and efficient primality proofs by use of computer algebra oracles. *Journal of Symbolic Computation*, 32(1/2):55–70, July 2001.
9. A. M. Cohen, H. Cuypers, and H. Sterk. *Algebra Interactive! Learning Algebra in an Exciting Way*. Springer-Verlag, 1999.
10. Y. Coscoy. A natural language explanation for formal proofs. In C. Retoré, editor, *Proceedings of LACL'96*, volume 1328 of *LNAI*, pages 149–167, Nancy, France, September 1996. Springer-Verlag.
11. Y. Coscoy, G. Kahn, and L. Théry. Extracting text from proof. In M. Dezani and G. Plotkin, editors, *Proceedings TLCA'95*, volume 902 of *LNCS*, pages 109–123. Springer-Verlag, April 1995.
12. E. Giménez. Codifying guarded definitions with recursive schemes. Technical Report 95-07, Ecole Normale Supérieure de Lyon, December 1994.
13. X. Huang and A. Fiedler. Presenting machine-found proofs. In M. A. McRobbie and J.K. Slaney, editors, *Proceedings of Conference on Automated Deduction (CADE-13)*, volume 1104 of *LNAI*, pages 221–225. Springer-Verlag, July/August 1996.
14. M. Kohlhase. OMDOC: Towards an internet standard for the administration, distribution, and teaching of mathematical knowledge. In J. A. Campbell and E. Roanes-Lozano, editors, *Proceedings of AISC 2000*, volume 1930 of *LNAI*, pages 32–52. Springer-Verlag, July 2000.
15. R. P. Nederpelt. Pseudo type theory: a lingua franca for mathematics (draft version), 2000. In preparation.
16. OMDOC: A Standard for Open Mathematical Documents. `http://www.mathweb.org/omdoc/index.html`.
17. M. Oostdijk. An interactive viewer for mathematical content based on type theory. Technical Report 00-15, Eindhoven University of Technology, September 2000.
18. OpenMath Consortium. The OpenMath Standard, August 1999. OpenMath Deliverable 1.3.3a.
19. OpenMath Society Website. `http://www.openmath.org`.
20. PolyMath Development Team. Java OpenMath Library: Version 0.7c. `http://pdg.cecm.sfu.ca/openmath/`, June 1999.
21. W3C XML Working Group. Extensible Markup Language (XML) 1.0. W3C Recommendation REC-xml-19980210, February 1998. Available at `http://www.w3.org/TR/1998/REC-xml-19980210`.