INFORMATION SOCIETY TECHNOLOGIES
(IST)
PROGRAMME

Project IST-2001-33562 MoWGLI

Prototype n. D2.b
Document Type Descriptors

Main Authors:
A. Asperti, M. Kohlhase, C. Sacerdoti Coen

Project Acronym: MoWGLI
Project full title: Mathematics On the Web: Get it by Logic and Interfaces
Proposal/Contract no.: IST-2001-33562 MoWGLI

Contents

1 Introduction

The representation of proofs is one of the central issues in the establishment of standard for mathematical documents on the Internet. The partners of the MoWGLI project have more than one representation format for proofs.

The HELM project exports mathematical knowledge from the Coq library and Coq proofs in the HELM CIC format (see report D2a), an Xml representation of terms in the Calculus of (Co)Inductive Constructions [Wer94], with some support for the Coq library structure. This representation is then transformed in several steps into a MathML presentation of the proof terms.

The OMDoc format (an Xml-based format for open mathematical documents; for OMDoc 1.1 see [Koh01]) defines a general content-based format for marking up proofs. This format supports content markup for proofs in a sequent-style proofs, mixing informal or natural language presentation with formal proof specification elements. In contrast to the HELM approach, which attempts to faithfully capture the CIC calculus in an Xml-based format, OMDoc strives for a general system-independent proof representation format in which specific proof formats can be embedded, and that is therefore suited to be a communication standard among systems.

This report comes out of the attempt to translate the HELM CIC format into OMDoc by XslT style sheets. The experiment and the resulting style sheets are covered in the companion document D2c of this report. These style sheets transform the the low-level XML description of the library of the Coq Proof Assistant to the version of OMDoc described in this report. Currently, the style sheets only cover a part of the ultimate transformation, covered by the original HELM format. This part consists in adding inner types (as content-MathML expressions) to the λ-terms exported from Coq and transforming the proof structure. The generation of natural language, line-breaking considerations, etc. will be implemented later in the task T2.5.

This experiment uncovered various deficiencies in the OMDoc format, which we try to remedy in this report. The main problems we address in this report are that

- OMDoc separates symbol declarations from definitions into two separate elements, which makes it impossible to represent mutually recursive definitions (without introducing specialized grouping devices). We will discuss an integrated notion of definitions in section 2.

- OMDoc proofs do not directly support the notion of proof contexts needed for reasoning from local hypotheses, e.g. in Gentzen's Natural Deduction calculus [Gen35]. It is therefore necessary to use a sequent-style encoding of hypotheses, which is awkward e.g. for theorem provers that employ the Curry-Howard-DeBruijn isomorphism.

  In section 3 we will introduce first-class proof contexts into the proof structures.

The OMDoc 1.1 format is currently being re-released as OMDoc 1.2 with a modularized document type definition and Xml schema. This prepares the way for a backward incompatible re-design of the modules in OMDoc 2.0. The extended representation format proposed in this report is intended as a basis for the re-design of the `PF` and `ST` modules in OMDoc 2.0.

2 Definitions and Axioms

In this section we will develop a new model of definitions and axioms in OMDoc.

Generally, mathematical formulas are built up from symbols whose meaning is given somewhere else in the document, or assumed to be known to the recipient. Therefore content-based representation formats for mathematical knowledge and formulae like MathML and OpenMath provide a notion of a *symbol*. Semantical representation formats like formal logics go even further by providing a notion of symbol (often called *constant* there) for all of the concepts (properties, functions, sets) involved formalizing the mathematical knowledge.

Declaring these symbols and defining or restricting their meaning is one of the foremost task of mathematical representation formats and knowledge management systems. The devices used for this are usually called **definitions** and **axiom**s in mathematics.

Before we come to the proposed OMDoc 2 model of definitions and axioms, let us look at some examples from informal mathematics and then survey the situation in OMDoc 1.

2.1 Informal Mathematical Data

There are many forms of axiomatic restrictions of meaning in mathematics. Maybe the best-known are the five Peano Axioms for natural numbers. Axioms are assertions about (sets) of mathematical objects and concepts that are assumed to be true.

1. 0 is a natural number.
2. The successor $s(n)$ of a natural number n is a natural number
3. 0 is not a successor of any natural number.
4. The successor function is one-one.
5. The set $\mathbb{N}$ of natural numbers contains only elements that can be constructed by axioms 1. and 2.

Figure 1: The five Peano Axioms

The Peano axioms in Figure 1 (implicitly) introduce three symbols: 0, the successor function s, and the set $\mathbb{N}$ of natural numbers. The five axioms jointly constrain their meaning so that any two structures that interpret 0, s, and $\mathbb{N}$ and satisfy these axioms must be isomorphic. Whenever this is the case, (and the axioms do not constrain the meaning of any other symbols), then a (set of) axioms is called a **definition**.

Generally, we will call the symbols to be defined the **definienda**, and the material a symbol is defined by the **definiens**.

Definitions can have many forms, they can be

1. equations, where the left hand side is the defined symbol and the left hand side is a term that does not contain it, as in Figure 2.1. We call such definitions **simple**.
2. general statements that uniquely determine the meaning of the objects or concepts in question, as in Figure 2.2. We call such definitions **implicit**; the Peano axioms are another example of this category.

1. $1 := s(0)$ (1 is the successor of 0)
2. the exponential function $e^{\cdot}$ is the solution to the differential equation $\partial f = f$ [where $f(0) = 1$].
3. addition on the natural numbers is defined by the following equations:

$$x + 0 = x \qquad \text{and} \qquad x + s(y) = s(x + y)$$

4. a natural number is called **even**, if it is the successor of an odd number, and **odd**, iff it is the successor of an even number; 0 is an even number.

Figure 2: Some common definitions

Note that this kind of definitions requires a proof of unique existence to ensure well-definedness. Incidentally, if we leave out the part in square brackets in Figure 2.2, the differential equation only characterizes the exponential function up to additive real constants. In this case, the "definition" only restricts the meaning of $e^{\cdot}$ to a set of possible values. In this case, we call such a "definition" **loose**.

3. sets of equations, as in Figure 2.3, even though this is strictly an implicit definition, it is a sub-case, where well-definedness can be shown by giving an argument why the systematic applications of these equations terminates, e.g. by exhibiting an ordering that makes the left hand sides strictly smaller than the right-hand sides. We call we call such a definition **recursive**.

4. sets of mutually recursive statements, where the definiens of a symbol can mention the others. We call such a definition **mutually recursive**, since the well-definedness proof only needs to exhibit an ordering, but the definition can not be separated into self-contained definitions for the respective symbols involved.

2.2 The State in OMDoc Version 1

OMDoc 1 separates symbol declarations from definitions into two separate elements. For instance, to define a symbol for the number one from the number zero and the successor function, we need to declare the symbol in a `symbol` element and then define its meaning in a `definition` element, as in Listing 1.

Listing 1: An OMDoc 1 definition of the number one

```
<symbol id="one">
  <type system="STS"><OMS cd="setname1" name="N"/></type>
</symbol>

<definition id="one-def" for="one" type="simple">
  <CMP>1 is the successor of 0</CMP>
  <OMOBJ><OMA><OMS cd="nat" name="suc"/></OMS cd="nat" name="zero"/></OMA></OMOBJ>
</definition>
```

While this gives great flexibility for encoding informal mathematics – e.g. there may be no definition for a declared symbol, or more than one (equivalent) definitions, it becomes hard to

enforce structural properties of symbols like their scope. A related but distinct problem is to determine which definitions are in the scope when we give a new definition. OMDoc 1 solves part of these problems by an explicit management of scope by `theory` elements, but this is not sufficient, since theories do not nest in OMDoc 1 and moreover, they cannot be used to specify scoping inside mathematical objects like proofs.

Finally, it becomes hard to represent mutually recursive definitions (without introducing specialized grouping devices). Again, OMDoc 1 has a partial solution in the `adt` element (for so-called abstract data types), but the experiment of translating HELM proof representations into OMDoc has shown that this is not expressive enough.

2.3 Symbol-Declaring Elements

The prototypical symbol-declaring element in OMDoc is the `declaration` element. It declares a symbol whose meaning can later be clarified or restricted by `axiom`s. It shares a lot of the the structure with other symbol-declaring elements, therefore we introduce its content model in this section together with these. In contrast to the elements introduced in sections 2.4 and 2.5 (we call these *definitional*, since they unambiguously define the meaning of the symbols), it does not claim to fix the meaning of the declared symbol up to isomorphism.

Ontologically, OMDoc **definitions** distinguish **types** and **objects**. The former are special constructs denoting sets that can be used to describe other objects, and are used like type construct in modern programming languages. Consequently, OMDoc has two major definitional elements: `definition` and `sortdef`.

The elements `definition` and `sortdef` – as well as their children (`constructor`, `selector`, and `recognizer`) – implicitly declare **symbol**s that can be used elsewhere (see the discussion above). We call these elements **symbol-declaring**.

`declare` and the definitional OMDoc elements declare **symbol**s that can be used elsewhere in OMDoc documents. We call these elements **symbol-declaring**. The declared symbols are uniquely identified by the `name` attribute (the name of the symbol) and the `id` of the enclosing `theory` element (the theory of the symbol).

Technically, this information (the theory identifier and the `name`) is sufficient to allow referring back to this symbol as an OpenMath `OMS` or a MathML `csymbol` element. For instance the `definition` declaration in Listing 4 gives rise to a symbol that can be referenced as

- `<OMS cd="nat" name="one"/>` in OpenMath and as
- `<csymbol DefinitionURL="http://base.uri/#byctx(one@nat)"/>` in C-MathML[1].

To specify additional information about symbols, symbol-declaring elements can have groups of `commonname` and `type` elements as children.

Multilingual `commonname` elements are indexed by their `xml:lang` attribute and specify keywords: If the document containing the symbol-declaring element was stored in a data base system, the symbol could be looked up by the strings specified in the `commonname` children.

The `type` element groups and specifies type information. As there are many type and sort disciplines around, the `type` element carries a `system` attribute that specifies the name of the type system. Its value is usually the name of a theory that specifies a logical system or type

[1] MathML uses a URI reference for identifying symbols; in this case, we use the OMDoc "byctx" fragment identifier; for details see [Koh01].

system. It is not an error to have more than one `type` declaration per `system` attribute in a symbol-declaring element, this just means that the object has more than one type in the respective type/sort system. The content is a mathematical object of the type system specified in the `system` attribute.

Note that the symbol declaration only fixes the identity (and referening) of the declared symbol so far, but not common notations. Therefore, OMDoc allows to embed `presentation` elements that specify notations for symbols into symbol-declaring elements. Since specifiation of notation is not the focus of this report, we refer the reader to the OMDoc specification [Koh01] for details.

To get a feeling for this let us consider the declaration in Listing 2, a very simple formalization of the declaration in Figure 1.. The `type` elements specify that 0 is a positive number in the type system "`simple-sorts`" (whatever that may be) and a natural number in OpenMath's small type system [Dav99]. The `presentation` specifies the default presentation of the symbol `zero`, in this case, it is just the digit 0.

Listing 2: A `definition` of the natural number 1 from 0 and the successor function

```
<declaration id="zero-dec" name="zero">
  <CMP><OMOBJ><OMS cd="nat" name="zero"/></OMOBJ> is a natural number.</CMP>
  <type system="simple-sorts"><OMOBJ><OMS cd="nat" name="nat"/></OMOBJ></type>
  <type system="http://mbase.mathweb.org/omstd#sts">
    <OMOBJ><OMS cd="setname" name="N"/></OMOBJ>
  </type>
  <presentation><use format="default">0</use></presentation>
</declaration>
```

Note that the meaning of the symbol 0 is only partially determined by the information we have so far: we know that 0 is a natural number. In fact, we cannot fully determine the meaning of 0 in isolation, but we need all of the information given in Figure 1. Therefore, OMDoc allows to restrict the meaning of symbols by the formulae in `axiom` elements (see [Koh01] for details). In the case of the five Peano axioms in Figure 1, we would formalize the second axiom in `declaration` like the one in Listing 2 and the other three as `axioms`.

Note that the `type` elements in our example are also axiomatic in character, they restrict the meaning of 0 to be a member of a certain set ($\mathbb{N}$), and we could have replaced them by a suitable axiom about 0. Note that in such cases, wher an axiom directly pertains one distinguished symbol, the `axiom` element can be embedded inside the respective declaration (like the `type` elements in our example).

2.4 Defining Sets with Constructors by `sortdef`

We will use Listing 3 as a running example for the `sortdef` element. A `sortdef` element is a highly condensed piece of syntax that declares a **sort** (an inductively defined set, i.e. one that contains all objects that can be written by a set of constructors). As a consequence, a `sortdef` element contains a set of `constructor` and `insort` elements, the latter are a useful shorthand that allow to specify that all the constructors of a given sort can also be used. `insort` elements are empty, they only refer to a sort declared elsewhere in a `sortdef` with their `xref` attribute.

The `constructor` elements specify the OpenMath symbols that can be used to construct the elements of its sort. Since a constructor is in general an n-ary function, a `constructor` element contains n `argument` children that give the argument sorts of this function. Note that n may be 0 and thus the constructor element may be empty. Sometimes it is convenient to

specify the inverses of a constructor, for this OMDoc offers the possibility to add an empty `selector` element to an `argument`, its attribute `total` specifies whether this symbol is a total (value `'yes'`) or a partial (value `'no'`) function.

Finally, a `sortdef` element can contain a `recognizer` child that specifies a predicate (and OpenMath symbol) that is true, iff its argument is of the respective sort.

Listing 3: A `definition` for the natural numbers

```
<theory id="nat">
  <sortdef id="nat.sort" name="nat" semantics="free">
    <commonname>The set of natural numbers</commonname>
    <constructor name="zero"/>
    <insort xref="pos"/>
  </sortdef>

  <sortdef id="pos.sort" name="pos" semantics="free">
    <commonname>The set of positive natural numbers</commonname>
    <constructor name="succ">
      <commonname>The successor function</commonname>
      <argument sort="nat">
        <selector total="yes" name="pred">
          <commonname>The predecessor function</commonname>
        </selector>
      </argument>
      <recognizer name="positive"/>
    </constructor>
  </sortdef>
  ...
</theory>
```

In our example in Listing 3 the `sortdef` elements define two sorts `pos` and `nat` for the (positive) natural numbers. Positive numbers are generated by the successor function (which is a constructor) on the natural numbers (all positive naturals are successors). On `pos`, the inverse `pred` of `succ` is total. The set `nat` of all natural numbers is defined to be the union of `pos` and the constructor `zero`. Note the use of mutuality here: the sort definition for `nat` refers to that of `pos-nat` and vice versa.

Furthermore, note that these `sortdef` definitions in Listing 3 are equivalent to the well-known Peano Axioms in Figure 1: the first two specify the sorts constructors, the third and fourth exclude identities between constructor terms, while the induction axiom states that `nat` is generated by `zero` and `succ`. We have specified this equivalence in the `semantics` attribute in the `sortdef` element for `nat`. A sort (or abstract data type) is called **free**, iff there are no identities between constructor terms, i.e. if two objects represented by different constructor terms can never be equal. An example of a free abstract data type is the theory of natural numbers.

An example of a abstract data type that is not free is the theory of finite sets given by the constructors `emptyset` and `insert` since the set $\{a\}$ can be obtained by inserting a into the empty set an arbitrary (positive) number of times. This kind of abstract data type is called **generated**, since it only contains elements that are expressible in the constructors.

If a abstract data type is **loose**, then it may contain other elements together with the ones generated by the constructors. Thus the `semantics` attribute of the `sortdef` element can have the values `'free'` and `'generated'` and `'loose'`. Note that the set of axioms equivalent to a `'generated'` sort would not contain axioms 3. and 4., and that for a sort with `'loose'` `semantics` would also lack the analogue of the induction axiom (5.).

2.5 Defining Non-Constructor Symbols

Objects that are not constructors can be defined by the `definition` in OMDoc. Since it is a symbol-declaring element, it has the content model described in section 2.3. It additionaly allows one of two primary elements that specify the meaning of the respective new symbol: an

`OMOBJ/m:math/legacy` with an OpenMath representation of a logical formula. Here, the meaning of the definition is that the defined symbol can be replaced by the content of the `omobj` element.

`rule` with a set of **case equations** whose left and right hand sides are specified by two children of the `rule` element. Here the intended meaning of the defined symbol is that the content of the second child (with the variables suitably substituted) can be replaced for an instance of the first child.

their meaning is further specified by a `type` attribute which can have the values

`'implicit'` This kind of definition is often (more accurately) called definition by desription, since the definiendum is described so accurately, that there is exactly one object satisfying the description. Thus as in the `'loose'` case, the `definition` element does not contain a replacement definition as an `OMOBJ/m:math/legacy` element. However, the meaning of the symbol is fully specified by the `axiom` siblings of the `definition` element. To verify this, the dominating `definition` element must have a `well-defined` attribute that points to a well-definedness `assertion`. This states that for each symbol defined in the `definition` element there can be at most one object that satisfies the `axiom`s, and that such objects exist. We give an example of an implicit definition in Listing 6.

`'simple'` **(default)** in this case the replacement formula may not contain the symbol itself, or one whose definition depends on this one, since this would result in a cyclic replacement. Since well-definedness is trivial here, a `well-defined` attribute is not needed.

To get a feeling for this let us consider the definition in Listing 4, a very simple formalization of the definition in Figure 2.1. The `definition` is a simple definition for the symbol `one`. Its `CMP` child holds the natural language version of the definition, and the `FMP` holds the formal version of this. The `type` elements specify that 1 is a positive number in the type system "`simple-sorts`" (whatever that may be) and a natural number in OpenMath's small type system [Dav99].

`'recursive'` in contrast to the `'simple'` case, the substitution formulae can contain definitional cycles, e.g. the defined symbol itself. To guarantee termination of the recursive instantiation (this is necessary to ensure well-definedness), it is possible to give a **measure** (an OpenMath formula) that maps the arguments to values in a well-founded **ordering** in the optional `measure` and `ordering` element. Alternatively, a termination proof can be specified as part of the `well-defined` attribute of the dominating `definition` element.

`'obj'` this can be used to directly give the concept defined here as an OpenMath object, e.g. as a group representation generated by a computer algebra system.

Listing 4: A `definition` of the natural number 1 from 0 and the successor function

```
<definition id="one-def" name="one" type="simple">
  <CMP>1 is the successor of 0</CMP>
  <type system="simple-sorts"><OMOBJ><OMS cd="nat" name="pos"/></OMOBJ></type>
  <type system="http://mbase.mathweb.org/omstd#sts">
    <OMOBJ><OMS cd="setname" name="N"/></OMOBJ>
  </type>
  <FMP>
    <OMOBJ><OMA><OMS cd="nat" name="succ"/><OMS cd="nat" name="zero"/></OMA></OMOBJ>
  </FMP>
</definition>
```

Listing 5: A recursive `definition`

```
<definition id="plus-def" name="plus" type="recursive">
  <commonname>addition</commonname>
  <CMP>Addition is defined by recursion on the second argument</CMP>
  <rule>
    <OMOBJ>
      <OMA><OMS cd="nat" name="plus"/><OMV name="X"/><OMS cd="nat" name="zero"/></OMA>
    </OMOBJ>
    <OMOBJ><OMV name="X"/></OMOBJ>
  </rule>
  <rule>
    <OMOBJ>
      <OMA><OMS cd="nat" name="plus"/>
        <OMV name="X"/>
        <OMA><OMS cd="nat" name="succ"/><OMV name="Y"/></OMA>
      </OMA>
    </OMOBJ>
    <OMOBJ>
      <OMA><OMS cd="nat" name="succ"/>
        <OMA><OMS cd="nat" name="plus"/><OMV name="X"/><OMV name="Y"/></OMA>
      </OMA>
    </OMOBJ>
  </rule>
</definition>
```

Listing 6: An implicit definition of the exponential function

```
<definition id="exp" type="implicit" well-defined="exp-well-def"/>
  <FMP><+exp' = exp ∧ exp(0) = 1+></FMP>
</definition>
<assertion id="exp-well-def">
  <CMP>
    There is at most one differentiable function that solves the
     differential equation in Definition <ref id="exp" xref="exp-def"/>.
  </CMP>
</assertion>
```

2.6 Mutually (recursive) Definitions

In general, definitions can be more general than the simple example in Listing 4 above, in Figure 2.4 we have seen a mutually recursive definition. For these situations, OMDoc uses the `mutual` element[2], whose content is a list of

`declaration` to declare new symbols and

[2]This element is similar to the `joint` element used to group proofs into groups for joint inductions; see section 3.

`sortdef` elements for defining **sorts**, i.e sets build populated by objects built up from **constructor** symbols and

`definition` elements for defining additional symbols in terms of constructors.

The `mutual` element carries an `id` for identification and an attribute `type` that can have the values `'recursive'` and `'corecursive'` Every element of a recursively defined set is a finite tree whose nodes are constructors. Typical examples of recursive definitions are natural numbers (constructed by 0 and successor), lists (constructed by the empty list and the `cons` operator), binary trees (constructed by the empty tree and the `node` operator). A function whose domain is a recursively defined type/set is defined by *structural recursion* if the definition is recursive and every recursive call is applied to a subterm of the term. For example, structural recursion over natural numbers allows recursion only over the predecessor, while structural recursion over binary trees allows recursive calls only when applied to the two sub-trees. Since every recursive element is a finite tree, every function defined by structural recursion is total and every definition given by structural recursion is well-founded.

Dually, every element of a co-recursively defined set is a (possibly) infinite tree whose node are constructors. The canonical example of co-recursive definitions is a stream (i.e. an infinite list). Co-recursive definitions are especially useful in the study of infinite processes, such as process algebras in computer science. A function whose codomain is a co-recursively defined type/set is defined by *structural co-recursion* if the definition is recursive and every recursive call is the argument of a constructor. Thus co-recursive functions can build infinite objects. The canonical example of a co-recursive function is the function that builds the stream of integer numbers $[0; 1; 2; \ldots]$: the function increments the counter and build the stream whose head is the counter value and whose tail is built by the recursive call. Since every functions whose domain is co-recursively defined can not be recursive, it must be able to inspect only a finite prefix of its input. Thus, in a lazy setting, every function is total and every co-recursive definition enjoys a property, dual to the one of well-foundedness, that ensures the correctness of the definition.

Every `mutual` element also carries an optional `well-defined` attribute that points to a well-definedness assertion and proof. Finally, the `mutual` element carries an optional attribute `local` which contains a whitespace-separated list of symbol names that are local (invisible outside) to the group of definition schemata inside the `mutual` element. If the attribute `local` is not present or the empty string, then all of the symbols declared inside globally visible. In this way, the `mutual` element restricts the visibility of symbols that are declared inside.

Listing 7: An Mutually Recursive Definition of “even” and “odd”.

```
<mutual id="even-odd-def" exports="even odd">
  <definition id="even-def" name="even">
    <CMP>
      A natural number is called <with class="definiens">even</with>,
      if it is the successor of an odd number.
    </CMP>
  </definition>
  <definition id="odd-def" name="odd">
    <CMP>
      A natural number is called <with class="definiens">odd</with>,
      if it is the successor of an odd number.
    </CMP>
  </definition>
</mutual>
```

3 Representing Proofs in OMDoc

Proofs form an essential part of mathematics and modern sciences. Conceptually, a proof is a representation of uncontroversial evidence for the truth of an assertion.

The question of what exactly constitutes a proof has been controversially discussed. The clearest (and most radical) definition is given by theoretical logic, where a proof is a sequence, or tree, or directed acyclic graph (DAG) of applications of inference rules from a formally defined logical calculus, that meets a certain set of well-formedness conditions. There is a whole zoo of logical calculi that are optimized for various applications. They have in common that they are extremely explicit and verbose, and that the proofs even for simple theorems can become very large. The advantage of having formal and fully explicit proofs is that they can be very easily verified, even by simple computer programs.

OMDoc provides the `proofobject` element for this; we will not go into this in this report, but concentrate on representing mathematical practice, where the notion of a proof is more flexible, and more geared for consumption by humans: any line of argumentation is considered as a proof, if it convinces its readers that it can be expanded to a formal proof in the sense given above. As the expansion process is extremely tedious, this option is very seldom really carried out explicitly in practice. Moreover, as proofs are geared towards communication among humans, they are given at vastly differing levels of abstraction. From a very informal proof idea to the initiated specialist of the field, who can fill in the details himself, down to a very detailed account for skeptics or novices. Note that such a proof will normally be still well above the formal level. Furthermore, proofs will normally be tailored to the specific characteristics of the audience, who may be specialists in one part of a proof while unfamiliar to the material in others. Typically such proofs have a sequence/tree/DAG-like structure, where the leaves are natural language sentences interspersed with mathematical formulae (often called "mathematical vernacular").

To provide a common markup system for mathematical practice of presenting proofs, OMDoc concentrates on the tree/DAG-like structure of proofs. It supports a proof format whose structural and formal elements are derived from hierarchical data structures developed for semi-automated theorem proving (satisfying the logical side), but which also allows natural language representations at every level (allowing for natural representation of mathematical vernacular at multiple levels of abstraction.) This proof representation (see [BCF+97] for a discussion and pointers) is a DAG of nodes which represent the proof steps. The proof steps contain a representation of the local claim and a justification by either a logical inference rule or higher-level evidence for the truth of the claim. This evidence can consist either of a proof method that can be used to prove the assertion, or by a separate subproof, that could be presented if the consumer was unconvinced. Conceptually, both possibilities are equivalent, since the method can be used to compute the subproof (called its expansion).

Expansions of nodes justified by method applications are computed, but the information about the method itself is not discarded in the process as in tactical theorem provers like Isabelle or NuPrL. Thus proof nodes may have justifications at multiple levels of abstraction in a hierarchical proof data structure. Note that the assertions in the nodes can be given as mathematical vernacular (in `CMP`s) or as logical formulae (in `FMP`s). This mixed representation enhances multi-modal proof presentation [Fie97], and the accumulation of proof information in one structure. Informal proofs can be formalized [Bau99]; formal proofs can be transformed to natural language [HF96]. The first is important, since it will be initially infeasible to totally formalize all mathematical proofs needed for the correctness management of the knowledge

base. Moreover, the hierarchical format allows to integrate various other proof representations like proof scripts (ΩMEGA replay files, ISABELLE proof scripts,...), references to published proofs, resolution proofs, etc, to enhance the coverage.

3.1 Informal Mathematical Data

To understand the content of a `proof` element, let us consider a proof (Figure 3) as it could be found in typical elementary textbook, only we have numbered the proof steps for referencing convenience.

Theorem: *There are infinitely many prime numbers.*
Proof: We need to prove that the set P of all prime numbers is not finite.

1. We proceed by assuming that P is finite and reaching a contradiction.
2. Let P be $\{p_1, \ldots, p_n\}$.
3. Let $q \stackrel{def}{=} p_1 \cdots p_n + 1$
4. Since for each $p_i \in P$ we have $q > p_i$,
5. we conclude $q \notin P$.
6. We prove the absurdity by showing that q is prime:
7. since for each $p_i \in P$ we have $q = p_i k + 1$ for a given natural number k, p_i can not divide q;
8. thus, since P is the set of all prime numbers, q must be prime. Q.E.D.

Figure 3: A proof example.

Even if the proof is very short and simple, we can observe in it several characteristics of a typical mathematical proofs. The proof starts with the thesis that is followed by eight main "steps" (numbered from 1 to 8). Some of the steps (2, 3, 4, 5, 7, 8) leave the thesis unmodified; these are called *forward reasoning* or *bottom-up* steps, since they are used to derive new knowledge from the available one, with the aim of reaching the conclusion. Some other steps (1, 6) are used to conclude the (current) thesis by opening new subproofs, each one characterized with a new local thesis. These steps are called *backward reasoning* or *top-down* steps, since they are used to reduce a complex problem (proving the thesis) to several simpler problems (the subproofs). In our example, both backward reasoning steps open just a new subproof: step 1 reduces the goal to proving that the finiteness of P implies a contradiction; step 6 reduces the goal to proving that q is prime. Note that we consider the last step (line 8) as a forward step that concludes exactly the thesis; it is perfectly reasonable to consider it also as a special case of backward reasoning step, reducing the problem to an empty set of new tasks. An example of backward reasoning step that opens more that just one subproof is the usual reasoning by induction over natural numbers, where we reduce the proof of a property $P(n)$ to the two tasks of proving $P(0)$ and proving $P(m+1)$ under the assumption that $P(m)$ holds. There may be steps that play both roles, being at the same time backward reasoning and forward reasoning steps; nevertheless, their usage in mathematical textbooks is very limited.

Some of the steps (2 and 3) are used to introduce new hypotheses, local declarations or local definitions, whose scope extends from the point were they are introduced to the end of the current subproof, covering also all the subproofs that are introduced. In our example, the

scope of the hypothesis that $P = \{p_1, \ldots, p_n\}$ (line 2) are the lines 3-8, while the scope of the local definition of q (line 3) are the lines 4-8. In an inductive proof, for example, the scope of the inductive hypothesis covers only the proof of the inductive step and not the proof of the base case (independently from the order adopted to present them to the user).

The difference between an hypothesis and a local declaration is that the latter is used to introduce as a variable a new element in a given set or type; the former, instead, is used to locally state some property of the variables in scope. For example, *"let n be a natural number"* is a declaration, while *"suppose n to be a multiple of 2"* is an hypothesis. The introduction of a new hypothesis or local declaration should always be justified by a proof step that discharges it. In our example the declaration P is discharged by step 1. Local definitions, instead, need no discharging.

To sum up, every proof step is characterized by a current thesis and a *context*, which is the set of all the local declarations, hypotheses and local definitions in scope. Furthermore, a step can either introduce a new hypothesis, definition or declaration or can just be a forward or backward reasoning step (called, from now on, a *derive* step). It is a forward reasoning derive step if it leaves the current thesis as it is. It is a backward reasoning derive step if it opens new subproofs, each one characterized by a new thesis and, possibly, also a new context.

3.2 Content Markup for Mathematical Proofs

Let us now come to the concrete markup scheme for proofs provided by OMDoc. Due to the complex hierarchical structure of proofs, sometimes it can be infeasible to utilize the tree-like structure provided by XML; for these cases we use cross-referencing. Proofs are specified by `proof` elements in OMDoc that have the attributes `id`, `for`, and `theory`. The `for` attribute points to the assertion that is justified by this proof (this can be an `assertion` element or a `derive` proof step, thereby making it possible to specify expansions of justifications and thus hierarchical proofs). Note that there can be more than one proof for a given assertion.

The `proof` element starts with an optional `metadata` element followed by an optional `label` element and it is a sequence of the following elements, with the constraint that it must end in a `derive` or `omtext` element, which is the (logical) root of the proof.

`hypothesis` elements allow to specify local assumptions, well-known from calculi like Gentzen's Natural Deduction calculus [Gen35]. They allow the hypothetical reasoning discipline needed for instance to specify proof by contradiction, by case analysis, or simply to show that A implies B, by assuming A and then deriving B from this local hypothesis. The scope of an hypothesis extends till the end of the `proof` element containing it. An important special case of hypothesis is the case of so-called "inductive hypothesis", this can be flagged by setting the value of the attribute `inductive` to `'yes'`; the default value is `'no'`. Example: *"let S be finite"*.

`declaration/definition/sortdefmutual` These elements are similar to the `hypothesis` elements, but they are used to introduce new local symbols. They are described in section 2. Example: *"let S be a set of integers"*.

`joint` elements are used to group together a piece of proof context. Their content model differs from the one for `proof` in that they allow `assertion` and `proof` elements but no `derive` elements. The optional `type` attribute can be used to determine the scoping rules that are applied *inside* the `joint` element. The legal values are `'inductive'`

and `'coinductive'`. When the attribute value is `'inductive'` and inside the `joint` element there are `assertion` elements together with their `proof` elements, the assertions can be recursively used in their proof. Additional conditions can be used to establish the well-foundedness of the proof[3]. The usual induction principle over natural numbers, for example, can be proved by well-founded induction in logical systems allowing this primitive. `'coinductive'` is used in the dual case of proof by co-induction.

`derive` elements are used both for backward-reasoning steps or for forward-reasoning steps. They derive a new claim from already known ones, from assertions or axioms in the current theory, from the assumptions of the assertion that is under consideration in the proof or from the new claims that they introduce in their subproofs (in case of top-down steps). We will explain the `derive` elements content in detail below.

`omtext` elements are used for explanatory text that can be freely intermixed with the other elements.

The most versatile element that can appear inside a proof is the `derive` element. Since we want to use the `derive` element for several different kinds of proof steps (informal description, sequent calculus rule application, natural deduction introduction or elimination rule application, tactics application), we need a high degree of flexibility. That flexibility is given by the `method` element that is used as the justification of the `derive` step: the attribute `xref` is a reference to a method description, to be (either formally or informally) defined somewhere in an external content dictionary (CD). The arguments of the method are given as children of the `method`. Several kind of arguments are allowed:

`premise` A premise is a reference either to an `hypothesis` in scope or to a `derive` step in the proof context (i.e. in a `proof` element that is an ancestor of the `premise`). The actual reference is specified using the `xref` attribute in both cases; the attribute contains a URI that points to the target.

`lemma` A `lemma` is a reference (via the `xref` attribute; see above) to an external `assertion` or `axiom` that is used as a lemma in the proof.

Note that the referenced `assertion` or `axiom` must be in the scope of the current theory, i.e the theory referenced in the `theory`) attribute of the outermost `proof` element. An `axiom` or `assertion` element is in scope of the current theory, if it is inside a `theory` element (or in case of the `assertion` element references a theory in its `theory` attribute) that is imported (directly or indirectly) by the current theory.

Furthermore note that a proof containing a `lemma` element is not self-contained evidence for the validity of the `assertion` it proves. Of course it is only evidence for the validity at all (we call such a proof grounded), if all the `assertion` targets of `lemma` references have grounded proofs themselves and the reference relation does not contain cycles. A grounded proof can be made self-contained by replacing all `lemma` elements by their target `assertion`s together with a at least one `proof`[4]

`proof` Backward reasoning steps have `proof` elements holding their subproof. A derive step is a top-down step if it has at least one `proof` element. The prime example of this are

[3] In particular, the assertion must have at least an argument that decreases in each recursive applications.

[4] Note that the OMDoc format allows this replacement, since `assertion` and `proof` are allowed wherever `lemma` is.

inductive proofs, where the method is "by induction" and the proofs for the base and step cases are subproofs.

`OMOBJ/m:math/legacy` Any other method argument which is not a premise, a lemma, a subproof or another method are given as OpenMath, MathML, or OMDoc legacy format expressions. A typical usage example is to give the witness of an existential statement.

`method` As OMDoc proofs serve as a general markup language for informal and formal proofs, they must also be able to deal with so-called "proof scripts. Proof scripts are structured sequences of commands used in tactic-based proof-assistants to make the system proceed in the proof. They can be thought as a macro-language to interact with the system. The basic proof script commands are called tactics, they are programs that synthesize parts of a proof. In this respect tactics are natural candidates for method descriptions. Most proof assistants have formal inference rules as basic (primitive) tactics and compose tactics to more powerful ones by so-called tacticals[5] A tactical is a functor whose input is a list of tactics and whose output is a new tactic. To account for this use in proof scripts we must be able to encode application of tacticals.

Example: the `Then` tacticals has two arguments which are two tactics; it behaves as the composition of the first argument with the second (i.e. `(Then T1 T2)` applies the tactic `T1` to the current goal and then applies the tactic `T2` to the new goal generated by `T1`). In OMDoc the `method` element is used for the tactics arguments of a tactical. If, for instance the tactics `T1` and `T2` do not take arguments, the justification `(Then T1 T2)` can be encoded in OMDoc as follows

```
<method xref="#byctx(then@CoqTacticals)"><method xref="T1"/><method xref="T2"/></method>
```

[5]The expressive power of tactical languages has been progressively increased during the last years. For example, the scripting language of the Coq proof-assistant is now an almost fully-fledged programming language similar to the ML language.

Listing 8: An OMDoc representation of the proof in Figure 3.

```
<assertion id="a1"><CMP>There are infinite prime numbers</CMP></assertion>
<proof for="a1">
  <derive id="pd1">
    <CMP>There are infinite prime numbers</CMP>
    <method xref="By_definition_of">
      <OMS xref="infinite"/>
      <proof>
        <derive id="pd2">
          <CMP>If P is finite then absurd.</CMP>
          <method xref="Discharge">
            <proof>
              <hypothesis id="h1"><CMP>Let P be {p_1, ..., p_n}</CMP></hypothesis>
              <definition id="d1" name="q">
                <CMP>Let q =def p_1 ... p_n + 1</CMP>
              </definition>
              <derive id="p1">
                <CMP>For each p_i ∈ P we have q > p_i</CMP>
                <method xref="Trivial"><premise xref="d1"/></method>
              </derive>
              <derive id="p2">
                <CMP>q ∉ P</CMP>
                <method xref="Trivial"><premise xref="p1"/></method>
              </derive>
              <derive id="p3">
                <CMP>False</CMP>
                <method xref="Contradiction">
                  <premise xref="p2"/>
                  <proof>
                    <derive id="p3">
                      <CMP>For each p_i ∈ P we have q = p_i k + 1 for a given natural number k</CMP>
                      <method xref="By_Definition"><premise xref="d1"/></method>
                    </derive>
                    <derive id="p4"><CMP>Each p_i ∈ P does not divide q</CMP></derive>
                    <derive>
                      <CMP>q is prime</CMP>
                      <method xref="Trivial"><premise xref="h1"/><premise xref="p4"/></method>
                    </derive>
                  </proof>
                </method>
              </derive>
            </proof>
          </method>
        </derive>
      </proof>
    </method>
  </derive>
</proof>
```

A The Document Type Definition

We reprint the relevant parts of the new OMDoc document type definitions here.

A.1 DTD-Module ST: Mathematical Statements

The DTD module ST deals with mathematical statements like assertions and examples in OMDoc. The entities file introduces parameter entities for the top-level element names it can be referenced by the public identifier `"-//OMDoc//ENTITIES OMDoc ST V1.2//EN"`

```
<!--
  An XML DTD for Open Mathematical documents (OMDoc 1.9): Module ST Entities
    PUBLIC: -//OMDoc//ENTITIES OMDoc ST V1.9//EN
    SYSTEM: <mowgli-CVS>/omdocst.ent
  See the documentation and examples at http://www.mathweb.org/omdoc
  (c) 1999-2003 Michael Kohlhase, released under the GNU Public License (GPL)
-->

<!ENTITY % omdocst.theory.qname "%omdoc.pfx;theory">
<!ENTITY % omdocst.type.qname "%omdoc.pfx;type">
<!ENTITY % omdocst.imports.qname "%omdoc.pfx;imports">
<!ENTITY % omdocst.axiom.qname "%omdoc.pfx;axiom">
<!ENTITY % omdocst.definition.qname "%omdoc.pfx;definition">
<!ENTITY % omdocst.declaration.qname "%omdoc.pfx;declaration">
<!ENTITY % omdocst.assertion.qname "%omdoc.pfx;assertion">
<!ENTITY % omdocst.example.qname "%omdoc.pfx;example">
<!ENTITY % omdocst.sortdef.qname "%omdoc.pfx;sortdef">
<!ENTITY % omdocst.mutual.qname "%omdoc.pfx;mutual">

<!ENTITY % omdocst.class
           "|%omdocst.type.qname;
            |%omdocst.assertion.qname;
            |%omdocst.example.qname;">

<!ENTITY % omdocst.theory.class "|%omdocst.theory.qname;">

<!ENTITY % omdocst.def.class
           "%omdocst.definition.qname;
            |%omdocst.declaration.qname;
            |%omdocst.sortdef.qname;
            |%omdocst.mutual.qname">

<!ENTITY % omdocst.onlyintheory.class
           "%omdocst.def.class;
            |%omdocst.axiom.qname;
            |%omdocst.imports.qname;">
```

The ST elements are defined in the following file, which can be referenced by the public identifier `"-//OMDoc//ELEMENTS OMDoc ST V1.2//EN"`,

```
<!--
  An XML DTD for Open Mathematical documents (OMDoc 1.9) Module ST
    SYSTEM: <mowgli-CVS>/omdocst.dtd
    PUBLIC -//OMDoc//DTD OMDoc ST V1.9//EN
  See the documentation and examples at http://www.mathweb.org/omdoc
  (c) 1999-2003 Michael Kohlhase, released under the GNU Public License (GPL)
-->

<!-- qnames for omdoc statements -->
<!ENTITY % omdocst.rule.qname "%omdoc.pfx;rule">
<!ENTITY % omdocst.commonname.qname "%omdoc.pfx;commonname">
<!ENTITY % omdocst.measure.qname "%omdoc.pfx;measure">
<!ENTITY % omdocst.ordering.qname "%omdoc.pfx;ordering">
<!ENTITY % omdocst.requation.qname "%omdoc.pfx;requation">
```

```
<!ENTITY % omdocst.insort.qname "%omdoc.pfx;insort">
<!ENTITY % omdocst.constructor.qname "%omdoc.pfx;constructor">
<!ENTITY % omdocst.recognizer.qname "%omdoc.pfx;recognizer">
<!ENTITY % omdocst.argument.qname "%omdoc.pfx;argument">
<!ENTITY % omdocst.selector.qname "%omdoc.pfx;selector">

<!ENTITY % symbol.attrib "name CDATA #REQUIRED">

<!ENTITY % omdocst.intheory.class
            "%omdocst.onlyintheory.class;
             %omdocdoc.class;
             %omdocst.class;
             %omdocpf.class;
             %omdocpres.class;
             %omdocext.class;
             %omdocquiz.class;
             %omdoc.extra.class;
             %omdoccth.sth.extra;">

<!ENTITY % otherdefinitiontype "">
<!ENTITY % definitiontype "(implicit|simple|recursive|obj
                            %otherdefinitiontype;)">

<!ENTITY % mcct "%omdocdoc.meta.content;(%omdocmtxt.CMP.qname;|
                                         %omdocst.commonname.qname;|
                                         %omdocst.type.qname;
                                         %omdocpres.symbol.class;)*">
<!ENTITY % mcc "%omdocdoc.meta.content;(%omdocmtxt.CMP.qname;)*,
                (%omdocst.commonname.qname;%omdocpres.symbol.class;)*">
<!ENTITY % cfm "%omdocdoc.meta.content;
                (%omdocmtxt.CMP.qname;)*,(%omdocmtxt.FMP.qname;)?">

<!ENTITY % welldef.attrib "well-defined CDATA #IMPLIED">
<!-- the attribute 'well-defined' is an URIref that points to a proof
     of well-definedness of the definition -->

<!ELEMENT %omdocst.commonname.qname; (%omdoc.mtext.content;)*>
<!ATTLIST %omdocst.commonname.qname;
          %xml.lang.attrib;
          %mid.attrib;>

<!ELEMENT %omdocst.type.qname; ((%omdocmtxt.CMP.qname;)*,(%omdocmobj.class;))>
<!ATTLIST %omdocst.type.qname; %idi.attrib;
                for CDATA #IMPLIED
                system CDATA #REQUIRED>

<!ELEMENT %omdocst.axiom.qname; (%cfm;)>
<!ATTLIST %omdocst.axiom.qname; %idg.attrib;>

<!ELEMENT %omdocst.mutual.qname;
          (%omdocdoc.meta.content;
           (%omdocmtxt.CMP.qname;)*,
           (%omdocst.sortdef.qname;|
            %omdocst.definition.qname;|
            %omdocst.axiom.qname;)*)>
<!ATTLIST %omdocst.mutual.qname;
          type (recursive|corecursive) #IMPLIED
          local CDATA #IMPLIED
          %welldef.attrib;
          %id.attrib;>
<!-- attribute 'local is a whitespace-separated list of symbol names
     that are local (invisible outside). -->

<!ELEMENT %omdocst.measure.qname; (%omdocmobj.class;)>
<!ATTLIST %omdocst.measure.qname; %mid.attrib;>

<!ELEMENT %omdocst.ordering.qname; (%omdocmobj.class;)>
```

```
<!ATTLIST %omdocst.ordering.qname; %mid.attrib;>

<!ELEMENT %omdocst.assertion.qname; (%cfm;%omdocpf.opt.proof;)>
<!ATTLIST %omdocst.assertion.qname; %idg.attrib;
                        theory NMTOKEN #IMPLIED
                        type (%assertiontype;) "conjecture"
                        name CDATA #IMPLIED
                        evidence CDATA #IMPLIED>
<!-- the %assertiontype; has no formal meaning yet, it is solely
     for human consumption. The 'generated-by' is for
     theory-interpretations, which can generate assertions.
     'evidence is a list of URIRefs that points to proofs or
     counter-examples-->

<!ELEMENT %omdocst.example.qname;
            (%omdocdoc.meta.content;(%omdocst.definition.qname;)*,
             (%omdocmtxt.CMP.qname;)*,(%omdocmobj.class;)*)>
<!ATTLIST %omdocst.example.qname;
            %omdoc.xmlns.theory.attrib;
            type (for|against) #IMPLIED
            assertion CDATA #IMPLIED
            %idref.attrib;>
<!-- attributes assertion is an URIref -->

<!ELEMENT %omdocst.sortdef.qname;
            (%mcct;,(%omdocst.constructor.qname;|%omdocst.insort.qname;)*)>
<!ATTLIST %omdocst.sortdef.qname;
            %id.attrib; %symbol.attrib; %welldef.attrib;
            semantics (loose|generated|free) "loose">

<!-- Definitions contain CMPs and concept specifications. The latter come in two
     forms. Type definitions define types and their constructors. Object
     definitions define mathematical objects. Note that both can define mutually
     dependent sets of types and objects. The the types and objects can be
     reached under this name in the content dictionary for the theory the
     definition is placed in. -->

<!ELEMENT %omdocst.declaration.qname;
            (%mcct;,(%omdocst.axiom.qname;)*)>
<!ATTLIST %omdocst.declaration.qname;
            %id.attrib; %symbol.attrib; %welldef.attrib;>


<!ELEMENT %omdocst.definition.qname;
            (%mcct;,
             ((%omdocst.rule.qname;)+|%omdocmobj.class;|%omdocmtxt.FMP.qname;)?,
             (%omdocst.measure.qname;,(%omdocst.ordering.qname;)?)?)>
<!ATTLIST %omdocst.definition.qname;
            %id.attrib; %symbol.attrib; %welldef.attrib;
            type %definitiontype; "simple">

<!ELEMENT %omdocst.rule.qname;
            ((%omdocmobj.class;),(%omdocmobj.class;))>
<!ATTLIST %omdocst.rule.qname;
            %idi.attrib;>

<!ELEMENT %omdocst.insort.qname; EMPTY>
<!ATTLIST %omdocst.insort.qname; xref CDATA #REQUIRED>
<!-- 'xref' is a reference to a sort symbol element -->

<!ELEMENT %omdocst.constructor.qname;
    (%mcc;,(%omdocst.argument.qname;)*,(%omdocst.recognizer.qname;)?)>
<!ATTLIST %omdocst.constructor.qname; %symbol.attrib;>

<!ELEMENT %omdocst.recognizer.qname; (%mcc;)>
<!ATTLIST %omdocst.recognizer.qname; %symbol.attrib;>
```

```
<!ELEMENT %omdocst.argument.qname; (%omdocst.selector.qname;)?>
<!ATTLIST %omdocst.argument.qname;
          sort CDATA #REQUIRED>
<!-- sort is a reference to a sort symbol element -->

<!ELEMENT %omdocst.selector.qname; (%mcc;)*>
<!ATTLIST %omdocst.selector.qname;
          %symbol.attrib;
          total (yes|no) "no">

<!ELEMENT %omdocst.theory.qname;
      (%omdocdoc.meta.content;(%omdocst.commonname.qname;)*,
       (%omdocmtxt.CMP.qname;)*,(%omdocst.intheory.class;)*)>
<!ATTLIST %omdocst.theory.qname;
          %omdoc.xmlns.theory.attrib;
          id ID #REQUIRED
          style NMTOKEN #IMPLIED>
<!-- theory identifiers should be unique per document -->

<!ELEMENT %omdocst.imports.qname;
          ((%omdocmtxt.CMP.qname;)*%omdoccth.imports.mix;)>
<!ATTLIST %omdocst.imports.qname;
          %omdoc.xmlns.theory.attrib;
          %from.attrib;
          hiding CDATA #IMPLIED
          type (local|global) "global">
<!-- hiding is a list of references to symbol ids -->
```

A.2 DTD-Module PF: Proofs and Proof objects

The DTD module PF deals with mathematical argumentation and proofs in OMDoc.

The entities file introduces parameter entities for the top-level element names it can be referenced by the public identifier `"-//OMDoc//ENTITIES OMDoc PF V1.2//EN"`

```
<!--
  An XML DTD for Open Mathematical documents (OMDoc 1.9): Module PF Entities
    PUBLIC: -//OMDoc//ENTITIES OMDoc PF V1.9//EN
    SYSTEM: <mowgli-CVS>/omdocpf.ent
  See the documentation and examples at http://www.mathweb.org/omdoc
  (c) 1999-2003 Michael Kohlhase, released under the GNU Public License (GPL)
-->

<!ENTITY % omdocpf.proof.qname "%omdoc.pfx;proof">
<!ENTITY % omdocpf.proofobject.qname "%omdoc.pfx;proofobject">

<!-- set this entity to the empty string, when not importing module PF -->

<!ENTITY % omdocpf.class "|%omdocpf.proof.qname;
                          |%omdocpf.proofobject.qname;">
<!-- use this entity to add an optional proof,
     for example inside an assertion -->
<!ENTITY % omdocpf.opt.proof ",(%omdocpf.proof.qname;
                                |%omdocpf.proofobject.qname;)">
```

The PF elements are defined in the following file, which can be referenced by the public identifier `"-//OMDoc//ELEMENTS OMDoc PROOF V1.2//EN"`,

```
<!--
  An XML DTD for Open Mathematical documents (OMDoc 1.9) Module PF
     SYSTEM: <mowgli-CVS>/omdocpf.dtd
     PUBLIC: -//OMDoc//DTD OMDoc PF V1.9//EN
  See the documentation and examples at http://www.mathweb.org/omdoc
  (c) 1999-2003 Michael Kohlhase, released under the GNU Public License (GPL)
```

```
−−>

<!−− qnames for omdoc statements −−>
<!ENTITY % omdocpf.metacomment.qname "%omdoc.pfx;metacomment">
<!ENTITY % omdocpf.derive.qname "%omdoc.pfx;derive">
<!ENTITY % omdocpf.hypothesis.qname "%omdoc.pfx;hypothesis">
<!ENTITY % omdocpf.joint.qname "%omdoc.pfx;joint">
<!ENTITY % omdocpf.method.qname "%omdoc.pfx;method">
<!ENTITY % omdocpf.premise.qname "%omdoc.pfx;premise">
<!ENTITY % omdocpf.lemma.qname "%omdoc.pfx;lemma">
<!ENTITY % omdocpf.label.qname "%omdoc.pfx;label">

<!ENTITY % omdocpf.proof.core.content
            "%omdocst.def.class;
            |%omdocpf.hypothesis.qname;
            |%omdocpf.joint.qname;
            |%omdocmtxt.omtext.qname;">

<!ENTITY % omdocpf.proof.full.content
            "%omdocpf.proof.core.content;
            |%omdocpf.derive.qname;">

<!ENTITY % omdocpf.proof.restricted.content
            "%omdocpf.proof.core.content;
            |%omdocst.assertion.qname;
            |%omdocpf.proof.qname;">

<!ELEMENT %omdocpf.proof.qname;
          (%omdocdoc.meta.content;(%omdocpf.label.qname;)?,
           (%omdocmtxt.CMP.qname;)∗,
           (%omdocpf.proof.full.content;)∗)>
<!−− Constraint: the context must end with either a derive or a mxtext −−>
<!ATTLIST %omdocpf.proof.qname;
          %omdoc.xmlns.theory.attrib;
          type (inductive|coinductive) #IMPLIED
          theory NMTOKEN #IMPLIED
          %idrefi.attrib;>

<!ELEMENT %omdocpf.proofobject.qname;
          (%omdocdoc.meta.content;(%omdocpf.label.qname;)?,
           (%omdocmtxt.CMP.qname;)∗,(%omdocmobj.class;))>
<!ATTLIST %omdocpf.proofobject.qname;
          %omdoc.xmlns.theory.attrib;
          theory NMTOKEN #IMPLIED
          %idref.attrib;>

<!ENTITY % just.content
            "(%omdocpf.method.qname;)?,
             (%omdocpf.proof.qname;|%omdocpf.proofobject.qname;)?">
<!−− The optional proof or proofobject is to be intended as the
     explosion/explanation of the method. −−>

<!ELEMENT %omdocpf.derive.qname;
          ((%omdocpf.label.qname;)?,(%omdocmtxt.CMP.qname;)∗,
           (%omdocmtxt.FMP.qname;)∗,%just.content;)>
<!ATTLIST %omdocpf.derive.qname;
          %omdoc.xmlns.theory.attrib;
          %id.attrib;>
<!−− CSC: there are two possibilities here in the case we also need a context:
     1. we can add the context to the derive
     2. we do not change the DTD and we use a dummy "proof" method with the
        explosion (that is just the proof). Is this solution compatible with
        the above comment about just.content?
−−>

<!ELEMENT %omdocpf.label.qname; (%omdoc.mtext.content;)∗>
<!−− CSC: WARNING!!! There is also PCDATA here! −−>
```

```
<!ELEMENT %omdocpf.hypothesis.qname;
          ((%omdocpf.label.qname;)?,(%omdocmtxt.CMP.qname;)*,
           (%omdocmtxt.FMP.qname;)*)>
<!ATTLIST %omdocpf.hypothesis.qname;
          %omdoc.xmlns.theory.attrib;
          %id.attrib;
          inductive (yes|no) "no">

<!ELEMENT %omdocpf.method.qname;
          ((%omdocpf.method.qname;|%omdocmobj.class;|
            %omdocpf.lemma.qname;|%omdocpf.premise.qname;|
            %omdocpf.proof.qname;)*)>
<!ATTLIST %omdocpf.method.qname;
          %omdoc.xmlns.theory.attrib;
          %css.attrib;
          xref CDATA #REQUIRED
          ranks CDATA #IMPLIED>
<!-- 'xref' is a pointer to the omdoc:symbol defining the method -->
<!-- The ranks string is a white-space separated list of non-negative numbers.
     Each number specifies the rank of the corresponding premise.
     The rank of a premise specifies its importance in the
     inference rule. Rank 0 is a real premise,
     whereas positive rank signifies sideconditions of
     varying degree. -->
<!-- CSC: we also need to encode other information for each premise/proof
     (e.g. its arity). Since this attribute just holds presentational
     information, it seems reasonable to put it into the "style" attribute
     (the one used for CSS).
 -->

<!ELEMENT %omdocpf.premise.qname; (%omdoc.mtext.content;)*>
<!ATTLIST %omdocpf.premise.qname;
          xref CDATA #REQUIRED>

<!ELEMENT %omdocpf.lemma.qname; (%omdoc.mtext.content;)*>
<!ATTLIST %omdocpf.lemma.qname;
          xref CDATA #REQUIRED>

<!ELEMENT %omdocpf.joint.qname; ((%omdocpf.proof.restricted.content;)*)>
<!ATTLIST %omdocpf.joint.qname;
          type (inductive|coinductive|
                inductive-recursive|coinductive-corecursive) #IMPLIED>
```

References

[Bau99] Judith Baur. Syntax und Semantik mathematischer Texte — ein Prototyp. Master's thesis, Saarland University, 1999.

[BCF+97] C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, K. Konrad, E. Melis, A. Meier, W. Schaarschmidt, J. Siekmann, and V. Sorge. ΩMEGA: Towards a mathematical assistant. In William McCune, editor, *Proceedings of the 14th Conference on Automated Deduction*, number 1249 in LNAI, pages 252–255, Townsville, Australia, 1997. Springer Verlag.

[Dav99] James Davenport. A small openmath type system. Technical report, The OpenMath Esprit Project, 1999. available at `http://monet.nag.co.uk/cocoon/openmath/standard/sts.pdf`.

[Fie97] Armin Fiedler. Towards a proof explainer. In J. Siekmann, F. Pfenning, and X. Huang, editors, *Proceedings of the First International Workshop on Proof Transformation and Presentation*, pages 53–54, Schloss Dagstuhl Germany, 1997.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen I & II. *Mathematische Zeitschrift*, 39:176–210, 572–595, 1935.

[HF96] Xiaorong Huang and Armin Fiedler. Presenting machine-found proofs. In M.A. McRobbie and J.K. Slaney, editors, *Proceedings of the 13th Conference on Automated Deduction*, number 1104 in LNAI, pages 221–225, New Brunswick, NJ, USA, 1996. Springer Verlag.

[Koh01] Michael Kohlhase. OMDoc: An open markup format for mathematical documents (version 1.1), 2001. `http://www.mathweb.org/omdoc/omdoc.ps`.

[Wer94] B. Werner. *Une Thorie des Constructions Inductives*. PhD thesis, Universit Paris VII, Mai. 1994.

Index