Translating the Mizar Mathematical Library into OMDoc format

Mihnea Iancu, Michael Kohlhase, and Florian Rabe
Computer Science, Jacobs University Bremen
⟨first-initial⟩.⟨lastname⟩@jacobs-university.de

September 7, 2011

Abstract

The Mizar Mathematical Library is one of the largest libraries of formalized mathematics. Its language is highly optimized for authoring by humans. As in natural languages, the meaning of an expression is influenced by its (mathematical) context in a way that is natural to humans, but harder to specify for machine manipulation. Thus its custom file format can make the access to the library difficult. Indeed, the Mizar system itself is currently the only system that can fully operate on the Mizar library.

This paper presents a translation of the Mizar library into the OMDoc format (**O**pen **M**athematical **Doc**uments), an XML-based representation format for mathematical knowledge. OMDoc is geared towards machine support and interoperability by making formula structure and context dependencies explicit. Thus, the Mizar library becomes accessible for a wide range of OMDoc-based tools for formal mathematics and knowledge management.

Contents

1 Introduction

Mizar [TB85] is a representation format for mathematics that is close to mathematical vernacular used in publications. Mizar is also a formal system for completing and verifying proofs written in the Mizar language. The continual development of the Mizar system has resulted in a centrally maintained library of mathematics: the Mizar mathematical library (MML). The MML is a collection of Mizar articles: text-files that contain definitions, theorems, and proofs. Currently the MML (version 4.166.1132) contains over 1000 articles with over 50000 theorems and over 10000 definitions.[1] Introductory information on Mizar and the MML can be found in [TR99] and [Wie99]. For the rest of this paper, we assume that the reader is at least superficially familiar with these basic texts.

The Mizar language is based on Tarski-Grothendieck set theory [Try90] formalized in (unsorted) first-order logic.[2] In addition, Mizar provides a very expressive and flexible type system that features dependent types as well as predicate restrictions [Ban03]. The Mizar language — in particular the type system and the input syntax — are highly optimized for authoring by humans. Consider for instance the following theorem:

```
for A being set holds
  A is finite iff ex f being Function st rng f = A & dom f in omega
```

For a skilled mathematician this can be almost read and understood without Mizar-specific training. The downside of this is that the Mizar system is currently the only system that can fully operate on the Mizar library, and as a consequence, many feel that the Mizar library is locked up in a custom file format that excludes it from the methods and tools developed in the mathematical knowledge management (MKM) communities.

In this paper, we show how he have remedied this situation by describing and implementing a translation of the MML into the OMDoc/LF language. OMDoc (**O**pen **M**athematical **Doc**uments [Koh06]) is an XML-based representation format for mathematical knowledge geared towards making formula structure and context dependencies explicit for machine support. OMDoc is parametric in the underlying logical formalism, and we use its instantiation with the Edinburgh Logical Framework (LF, [HHP93]) to formally define the Mizar language.

This Mizar to OMDoc transformation completely rethinks the information architecture and indeed enhances the OMDoc language design in the process.

Our translation satisfies three requirements that are as indispensable for interoperability as they are hard to combine: (i) it preserves the human-oriented structure of Mizar expressions and declarations, e.g., the type system is not coded out; (ii) it uses only the generic representational infrastructure of a simple framework language (OMDoc/LF in our case) so that further processing is possible without hard-coding any idiosyncrasies of Mizar; (iii) the result can be verified based on a formal representation of the Mizar syntax and semantics in LF.

In this report we will present the architecture and results of the system we designed for the translation and discuss the difficulties and importance of the project. We begin with a introduction to the Mizar language and system in section 3. Then, in section 4, we describe the OMDoc side and introduce some background notions relevant to the translation. Section 5 details our OMDoc representation of the Mizar language which we use as a base for the translation of the MML. In section 6 we present our implementation both from a theoretical and practical perspective. Finally, in Sect. 7 we discuss the results and conclude.

2 Related Work

The motivations for a translation of the Mizar library are not particular to the OMDoc format, and it is therefore not very surprising that the work reported on in this paper is not the first attempt to translate the Mizar library.

[1] See http://mmlquery.mizar.org/ for up-to-date statistics.

[2] Technically, theorem schemes and the Fraenkel operator of Mizar slightly transcend first-order expressivity, but the language is first-order in style.

When processing Mizar text we have the choice between two levels of language: The **pattern-level** (presentation-level) language is the rich human-oriented external syntax in which Mizar articles are written; and the **constructor-level** (semantic level) is the machine-internal representation used in the Mizar system.

The Mizar project produces a hyper-linked, pretty-printed version of the assertions of a Mizar article for publication in the journal *Formalized Mathematics* [Ban06a] with its electronic counterpart the *Journal of Formalized Mathematics* [JFM]. This pattern-level translation generates a human-oriented presentation of Mizar articles, where formulae are presented in mathematical notation using LaTeX and parts of the Mizar logical language are verbalized.

There have been various early hand translations of selected Mizar articles for benchmarking automated theorem provers (see e.g. [DW97]). In 1997/8 Czeslaw Bylinski and Ingo Dahn translated the MML into a PROLOG syntax by extending the Mizar system with a custom (constructor-level) PROLOG generator. This was done independently of the main Mizar code base, and as a result soon desynchronized with it. A structure-preserving transformation of constructor-level Mizar to OMDoc has been also attempted in [BK07], but has remained partial. Constructor-level Mizar has been used for information retrieval purposes in the MMLQ system which provides a web interface to and a query language; see [Ban06b] for details.

Finally, Josef Urban has defined a custom XML format in [Urb06b] covering initially mainly the constructor-level, and modified Mizar to use this format internally. In this way it is ensured that this XML format cannot desynchronize from the (often rapid) Mizar development, and can be used as a faithful translation layer for external tools. At the same time, this means that the format has to be quite Mizar-oriented, and its use in generic MKM systems typically requires a translation layer. One such layer — the MPTP system [Urb06a] — has been developed for translation targeted at automated theorem proving systems based on the TPTP syntax. The MPTP translation is mostly concerned with semantics and formulas, the presentation layer is practically omitted there, and a custom proof translation was only added experimentally later in [US08].

The work presented here is analogous to the MPTP translation in that it uses the Mizar XML as an API suitable for implementing a translation to a general mathematical format, allowing a number of independently developed general tools to work with the MML.

3 Mizar

3.1 Preliminaries

Mizar [TB85] can refer to one of the following things:

- The Mizar Language is a formal language for representing mathematics designed to be as similar as possible to the mathematical vernacular.
- The Mizar System is the only implementation of the Mizar Language and provides a proof checker for Mizar source files.
- The Mizar Mathematical Library (MML) is a library consisting of Mizar articles that were checked by the Mizar System and in addition reviewed by the Library Committee of the MML. Currently the MML contains over 50000 theorems and over 9500 definitions.

Mizar is based on classical first-order logic extended with second-order axiom schemes and featuring a rich ontology of primitive mathematical objects, types, and proof principles.

Mizar basic language constructors are *types*, *terms* and *formulas*.

Mizar language features include *Justified Theorems*, *Definitions*, *Schemes*, *Notations*, *Registrations* and *Clusters*.

3.2 Justified Theorems

Justified theorems are propositions together with a proof of correctness. For example, one can prove that if a set is not the empty-set ({} in Mizar) then it has at least one element. In Mizar this can be written as a justified theorem (so it can be used later in other proofs):

```
theorem
        X <> {} implies ex x st x in X
        proof
```

where we omit the proof.

Definitions in Mizar can be used to introduce *predicates*, *functions*, *attributes*, *modes* and *structures*. Since the Mizar language is very complex it is difficult to give a general form of each kind of definition so we will instead give a general form of the most used definition for each kind and then discuss the other language features pertaining to definitions separately.

3.3 Function Definitions

Functors can be introduced using the general form:

definition *let* x_1 *be* ϑ_1, $\cdots$, *let* x_n *be* ϑ_n;
        func $(x_{i1}, x_{i2}, \cdots, x_{ik})\varphi(x_{j1}, x_{j2}, \cdots, x_{jl})$ -> $\vartheta(x_1, x_2, \cdots, x_n)$ *means* $\delta(x_1, x_2, \cdots, x_k, it)$;
end;

where x_i are the arguments (terms) of the definition and ϑ_i their respective types. φ is the name of the function being defined and x_{ik} are left side arguments while x_{jl} are the right side arguments. $\vartheta(x_1, x_2, \cdots, x_n)$ is the function's return type (with x_i as possible parameters). Finally $\delta(x_1, x_2, \cdots, x_k, it)$ represents the formula giving the meaning of the function where x_i are arguments together with *it* which is placeholder for the result of the function.

For example the big union of a set (union of all its elements) can be defined as:

```
definition
        let X be set ;
        func union X -> set means
        for x being set holds
        ( x in it iff ex Y being set st
        ( x in Y & Y in X ) );
end;
```

3.4 Predicate Definitions

The general form of the most used type of predicate definition is:

definition *let* x_1 *be* ϑ_1, $\cdots$, *let* x_n *be* ϑ_n;
        pred $x_{i1}, x_{i2}, \cdots, x_{ik} \Pi x_{j1}, x_{j2}, \cdots, x_{jl}$ *means* $\delta(x_1, x_2, \cdots, x_k)$;
end;

where x_i are arguments (terms) and ϑ_i their respective types. Π is the name of the predicate with x_{ik} being left side arguments and x_{jl} being right side arguments. Finally $\delta(x_1, \cdots, x_n, it)$ represents the formula that gives the predicate its truth value.

For example, the subset relation (c= in Mizar) can be defined using:

```
definition
        let X, Y be set ;
        pred X c= Y means
        for x being set st x in X holds x in Y;
end;
```

3.5 Attribute Definitions

Attribute definitions are of the following form:

definition *let* x_1 *be* ϑ_1, $\cdots$, *let* x_n *be* ϑ_n;
attr Δ -> $\vartheta(x_1, ..., x_n)$ *means* $\delta(x_1, ..., x_n, it)$;
end;

In the example above x_i are the arguments (terms) and ϑ_i their respective types. Δ is the symbol (name) of the attribute and $\vartheta(x_1, .., x_n)$ is a type dependent on $x_1, \cdots, x_n$ representing the type to which the attribute is applied. Finally $\delta(x_1, \cdots, x_n, it)$ represents a predicate that must hold for $x_1, \cdots, x_n$ and for *it* which is a Mizar notation representing the result of the current definition. In this case, it refers to the result of applying Δ to $\delta(x_1, \cdots, x_n, it)$.

For example *non-empty* is an attribute of the type *set* here $\delta(it)$ is:

```
ex x being set st x in it
```

where *it* represents a *non-empty set*.

3.6 Mode Definitions

Mode definitions are used to introduce new types, usually using attributes:

definition *let* x_1 *be* ϑ_1, $\cdots$, *let* x_n *be* ϑ_n;
mode ϑ *is* $\Delta_1(x_{\sigma_{1_1}}, \cdots, x_{\sigma_{1_{k_1}}})$ $\Delta_2(x_{\sigma_{2_1}}, \cdots, x_{\sigma_{2_{k_2}}})$ $\cdots$ $\Delta_n(x_{\sigma_{n_1}}, \cdots, x_{\sigma_{n_{k_n}}})$ ϑ_1;
end;

As above x_i are the arguments (terms) and ϑ_i their respective types. $\Delta_i(x_{\sigma_{i_1}}, \cdots, x_{\sigma_{i_{k_i}}})$ are attributes with arguments and ϑ_1 represents the mother type to which the attributes are applied to obtain the new type ϑ. Note that the new type ϑ must be non-empty.

As an example, one can use the attribute *non-empty* to construct a subtype of set using:

definition
mode DOMAIN *is* non-empty set
end;

3.7 Structure Definitions

Structure definitions follow the same pattern as the others but are more complicated. Internally a structure definition is expanded into several constructors: for the aggregate functor, the structure mode, the strict attribute and one for each selector. One can think of structures as Mizar's implementation of record types. The structure mode defines the new type for the structure which may extend another structure. The selectors represent the members of the structure (fields of a record type). Finally one can think of the aggregator functor as a constructor for a record type .

The general form of structure definitions is:

definition *let* x_1 *be* ϑ_1, $\cdots$, *let* x_n *be* ϑ_n;
struct ($\Xi_1(x_1, x_2, \cdots, x_n), \Xi_1(x_1, x_2, \cdots, x_n), \cdots, \Xi_k(x_1, x_2, \cdots, x_n)$)
Ξ(# υ_1 -> $\vartheta(x_1, x_2, \cdots, x_n)$, υ_2 -> $\vartheta(x_1, x_2, \cdots, x_n, \upsilon_1)$, $\cdots$, υ_l ->
$\vartheta(x_1, x_2, \cdots, x_n, \upsilon_1, \upsilon_2, \cdots, \upsilon_{l-1})$#);
end;

where x_i and $theta_i$ are the definition parameters and their types, Xi is the structure currently being defined, $\Xi_i(x_1, x_2, \cdots, x_n)$ are the structures (with parameters) that the Ξ extends and υ_i are the selectors (fields).

An example for a structure definition is:

definition
struct (1-sorted) 2-sorted(#carrier,carrier' -> set#);
end;

where the first selector (carrier) is "inherited" from 1-sorted.

3.8 Review of Definitions

In additions to the types describe above (function, predicate, mode, $\cdots$) definitions can be categorized using other relevant properties. For instance they may be categorized as *direct* or *indirect* depending on whether the meaning of the definiens is explicit or implicit.

Direct definitions are in essence abbreviations and may be used for functors, predicates and modes. They are usually introduced via "is" and the meaning is expressed as a term ,except for predicates where "means" is used and the meaning is a formula. For example, a direct function definition for the successor function could be:

definition *let* x *be* Nat;
    func $succ\, x$ -> Nat *is* $x + 1$;
end;

Here the function *succ* is defined as a term by applying the functor + to x and 1.

Indirect definition introduce a definiens by giving its meaning implicitly and may be used for functors, attributes and modes. For example, an indirect function definition for the succesor function could be:

definition *let* x *be* Nat;
    func $succ\, x$ -> Nat *means* $x + 1 = it$;
end;

In Mizar *it* is a placeholder for the result of the application of the current definiens, hence the meaning is implicit.

In this case it stands for the result of $succ(x)$ and, consequently, the function is defined as a formula.

Another way of categorizing definitions is by looking at case based definitions. In all the examples above the meaning of definitions was given by a single term/formula. However Mizar supports case based definitions which allow for definiens meanings of the form:

$case\ C_1 \to R_1,\ case\ C_2 \to R_2 \cdots\ case\ C_k \to R_k,\ else\ R$

where C_i are the cases with R_i their respective results and R is the default result (in case none of the cases hold). The default result may be omitted if a proof that it is not necessary (meaning that $C_1 \vee C_2 \vee \cdots \vee C_k$ always holds) is provided. Case based definitions are thus of two kinds, *complete* (when the cases give cover all possibilities and no default case is required) or *partial* when there is a default case. Simple definitions (with no cases) can thus be considered the degenerate case of partial definitions.

3.9 Schemes

Mizar extends the classical first-order logic with second-order variables. Schemes are second order sentences which take functors and predicates as arguments. They may have premises (assumptions of the schema) and must have a justification (proof). The general form of a scheme is:

scheme Φ {$\xi_1, \xi_2, \cdots, \xi_n$}:
    $\delta(\xi_1, \xi_2, \cdots, \xi_n)$ *provided*
    $\delta_1(\xi_1, \xi_2, \cdots, \xi_n)$ *and* $\delta_2(\xi_1, \xi_2, \cdots, \xi_n)$ *and* $\cdots$ *and* $\delta_k(\xi_1, \xi_2, \cdots, \xi_n)$
    justification;

where Φ is the current scheme being defined, ξ_i are the second order variables (functors or predicates), δ is the scheme sentence and δ_i are the scheme premises (assumptions).

An example scheme is induction which can be defined as:

scheme Ind {$P[Nat]$}:
    for k *holds* $P[k]$ *provided*
    $P[0]$ *and for* k *st* $P[k]$ *holds* $P[k+1]$
    justification;

3.10 Notations

Notations introduce a new name for a Mizar construct and may be synonymic or antonymic. For example one can denote *odd* as an antonymic notation of *even* (provided *even* is previously defined):

```
notation
        let i be Integer;
        antonym i is odd for i is even;
end;
```

3.11 Clusters and Registrations

Alongside modes, clusters represent another use of attributes. A *cluster* is a set of attributes and can be used in the creation of types. Since attributes can be seen as predicates on types, clusters can be seen as the conjunction of the the applied predicates. Since, in Mizar, types must be non-empty one must prove that such a conjunction is non-empty before it can be used. Such definitions are called *registrations*.

For example one can cluster even or odd for the type of integers. Below, the proofs are omitted for brevity, but they rely on giving 0 and 1 respectively as examples to prove non-emptiness.

```
registration
        cluster even for Integer;
        existence
        proof;

        cluster odd for Integer;
        existence
        proof;
end;
```

Note that Mizar language exists at two levels. The **pattern-level** language (presented in the examples above) is the rich input syntax in which Mizar articles are written. The **constructor-level** language is the unique internal representation used by the Mizar system's proving engine. The pattern-level language is richer, more expressive and human-oriented but in an effort to mirror conventional mathematical notations is also full of notation-conflicts and not suitable for semantics preserving machine translation. Consequently, we use the unambiguous constructor-level language (which is as a source for our translation. This means, we use the output of the Mizar processor, consisting mainly of XML files [Urb05], as the input. The structure and content of the files are described in detail in section 6

4 Representing Mathematics in OMDoc/MMT

4.1 OMDoc

OMDoc is a content markup format and data model for mathematical documents. It models mathematical content using three levels of abstraction:

Object Level: OMDoc uses OpenMath and MathML as established standards for the markup of *formulae*. Mizar types, terms and formulas correspond to this level.

Statement Level: OMDoc supplies original markup for explicitly representing the *declarations* and *assertions* in mathematical theories. Mizar definitions, theorems, schemes, notations, and registrations correspond to this level.

Theory Level: Finally OMDoc offers original markup that allows for clustering sets of statements into *theories* as well as specifying relations between them (inclusions, morphisms). Mizar articles and imports between them correspond to this level.

Core OMDoc concentrates on the structural relations between these mathematical concepts. It deliberately avoids fixing language primitives for them and abstracts from specific mathematical foundations. This is a crucial design choice that makes OMDoc a universal representation format while remaining manageably simple.

For the case of Mizar, this means that core OMDoc does not feature exact analogues to Mizar's sophisticated definition principles. Neither can it adequately represent the theoremhood property that defines the semantics of Mizar formulae. This is not surprising because these features are highly specific to the syntax and semantics of individual languages. The extension of core OMDoc with such language-specific features is the role of *pragmatic* OMDoc, which we will discuss next.

Core OMDoc uses a minimal set of conceptually orthogonal representational primitives, resulting in expressions with canonical structure, which simplifies the theoretical analysis and implementation of core OMDoc expressions (e.g., see [KRZ10]). **Pragmatic OMDoc**, on the other hand, strikes a balance between verbosity and formality. It permits introducing a complex representational infrastructure that provides both a formal semantics and is intuitive for humans. In particular, the semantics of these language-specific extensions is defined entirely within core OMDoc. Thus, OMDoc can provide multiple "pragmatic vocabularies" catering to different communities and their tastes.

OMDoc achieves a pragmatic object level by being parametric in the foundational framework in which the syntax and semantics of a language are formalized. More concretely, a logical framework — LF in our case — is described as an OMDoc theory. Then a logic — Mizar's first-order logic in our case — is defined as another OMDoc theory with *meta-theory* LF; this in turn serves as the meta-theory for the actual object language of interest — in our case Mizar's Tarski-Grothendieck set theory. Finally, individual Mizar articles are represented as (conservative) extensions of this theory. The respective meta-theory induces the pragmatic semantics of the object theories: In particular, the LF type system induces the definition of well-formedness and provability of Mizar formulae. For the details, we refer to the MMT fragment [RK11] of OMDoc for the general framework and to [IR11] for the formalization of Mizar in LF.

To achieve a pragmatic statement level, we make use of statement patterns, which were introduced recently in MMT by Fulya Horozal and the third author. A statement pattern introduces a new kind of statement together with concrete syntax for it. Moreover, it defines the semantics of these statements in terms of core OMDoc. For example, standard first-order logic is defined using three patterns for function symbols, predicate symbols, and axioms/theorems, respectively. A pattern can have free variables, e.g., for the arity of a function symbol; and specific instances of a pattern must provide substitutions for these free variables. To define Mizar, we need to add several sophisticated patterns, e.g., we need a single pattern for case-based implicit function symbol definitions. This way, we are able to give a formalization of the syntax and semantics of both the object level and statement level of Mizar. [3] Effectively, we are able to recover a fragment of OMDoc that is isomorphic to Mizar.

4.2 MMT

MMT [RK08] is A Module System For Mathematical Theories. The MMT language is designed to be a fully formal sublanguage of the OMDoc format and focuses on foundation-independence, scalability and modularity. The MMT ontology is presented in Fig. 1.

MMT **modules** are of two kinds, **theories** and **views**. MMT theories consist of **symbol declarations** and MMT views consist of **symbol assignments**. **Constants** represent declarations of the base language and **structures** represent inheritance between theories. **Terms** appear inside MMT constants (e.g. as type or definition) and their grammar is motivated by the OpenMath grammar.

The MMT System provides an API to the MMT data structures described above which is used in our implementation. In fact this translation can be seen as an extension of MMT allowing for using Mizar files as input in addition to OMDoc files. This is discussed in more detail in section 6

[3] The corresponding pragmatic theory level is ongoing work, but not needed for Mizar.

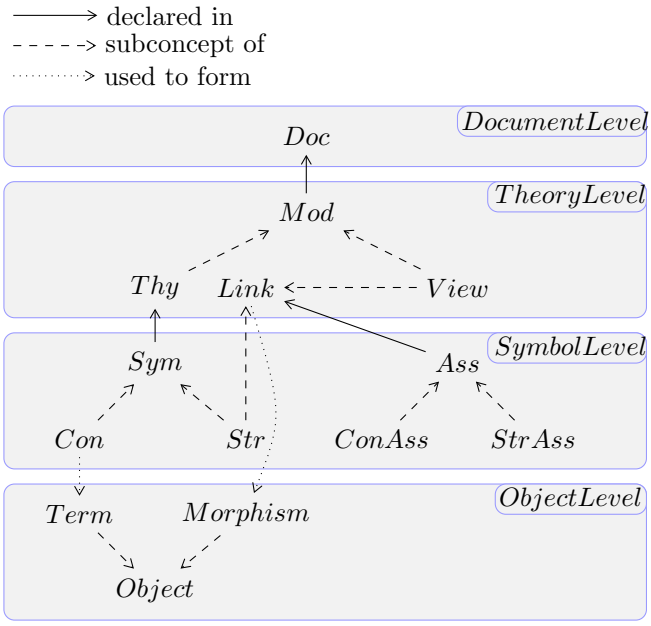

Figure 1: MMT Ontology

5 A Mizar Profile for OMDoc/MMT

Mizar is ontologically richer than OMDoc so we represent Mizar specific constructs by importing an OMODoc theory which encodes the Mizar grammar. We use LF as a language for writing this representation of the Mizar grammar and Twelf as an implementation of LF to proof check the encoding and export it to OMDoc.

5.1 Encoding of Mizar in LF

The Edinburgh Logical Framework [HHP93] (LF) is a formal meta-language used for the formalization of deductive systems. It is related to Martin-Löf type theory and the corner of the lambda cube that extends simple type theory with dependent function types and kinds. We will work with the Twelf [PS99] implementation of LF and its module system [RS09].

The central notion of the LF type theory is that of a *signature*, which is a list Σ of *kinded type family* symbols $a : K$ or *typed constant* symbols $c : A$. It is convenient to permit those to carry optional definitions, e.g., $c : A = t$ to define c as t. (For our purposes, it is sufficient to assume that these abbreviations are transparent to the underlying type theory, which avoids some technical complications. Of course, they are implemented more intelligently.)

LF *contexts* are lists Γ of typed variables $x : A$, i.e., there is no polymorphism. Relative to a signature Σ and a context Γ, the expressions of the LF type theory are *kinds* K, *kinded type families* $A : K$, and *typed terms* $t : A$. `type` is a special kind, and type families of kind `type` are called *types*.

We will use the concrete syntax of Twelf to represent expressions:

- The dependent function type $\Pi_{x:A} B(x)$ is written $\{x : A\}\, B\ x$, and correspondingly for dependent function kinds $\{x : A\}\, K\ x$. As usual we write $A \to B$ when x does not occur free in B.
- The corresponding λ-abstraction $\lambda_{x:A} t(x)$ is written $[x : A]\, t\ x$, and correspondingly for type families $[x : A]\, (B\ x)$.

- As usual, application is written as juxtaposition.

Given two signatures $\%\texttt{sig}\ S\ =\ \{\Sigma\}$ and $\%\texttt{sig}\ T\ =\ \{\Sigma'\}$, a *signature morphism* σ from S to T is a list of assignments $c := t$ and $a := A$. They are called *views* in Twelf and declared as $\%\texttt{view}\ v\ : S\ \to T\ = \{\sigma\}$. Such a view is well-formed if

- σ contains exactly one assignment for every symbol c or a that is declared in Σ without a definition,
- each assignment $c := t$ assigns to the Σ-symbol $c : A$ a Σ'-term t of type $\overline{\sigma}(A)$,
- each assignment $a := K$ assigns to the Σ-symbol $a : K$ a Σ'-type family K of type $\overline{\sigma}(K)$.

Here $\overline{\sigma}$ is the homomorphic extension of σ that maps all closed expressions over Σ to closed expressions over Σ', and we will write it simply as σ in the sequel. The central result about signature morphisms (see [HST94]) is that they preserve typing and $\alpha\beta\eta$-equality: Judgments $\vdash_\Sigma t : A$ imply judgments $\vdash_{\Sigma'} \sigma(t) : \sigma(A)$ and similarly for kinding judgments and equality.

Finally, the Twelf module system permits inclusions between signatures and views. If a signature T contains the declaration $\%\texttt{include}\ S$, then all symbols declared in (or included into) S are available in T via qualified names, e.g., c of S is available as $S.c$. Our inclusions will never introduce name clashes, and we will write c instead of $S.c$ for simplicity. Correspondingly, if S is included into T, and we have a view v from S to T', a view from T to T' may include v via the declaration $\%\texttt{include}\ v$.

This yields the following grammar for Twelf where gray color denotes optional parts.

Toplevel	G	::=	$\cdot \mid G,\ \%\texttt{sig}\ T\ =\ \{\Sigma\} \mid G,\ \%\texttt{view}\ v\ : S\ \to T\ = \{\sigma\}$
Signatures	Σ	::=	$\cdot \mid \Sigma,\ \%\texttt{include}\ S \mid \Sigma,\ c : A = t \mid \Sigma,\ a : K = t$
Morphisms	σ	::=	$\cdot \mid \Sigma,\ \%\texttt{include}\ v \mid \sigma,\ c := t \mid \sigma,\ a := A$
Kinds	K	::=	$\texttt{type} \mid \{x : A\}\, K$
Type families	A	::=	$a \mid A\ t \mid [x : A]\, A \mid \{x : A\}\, A$
Terms	t	::=	$c \mid t\ t \mid [x : A]\, t \mid x$

We will sometimes omit the type of a bound variable if it can be inferred from the context. Moreover, we will frequently use implicit arguments: If c is declared as $c : \{x : A\}\, B$ and the value of s in $c\ s$ can be inferred from the context, then c may alternatively be declared as $c : B$ (with a free variable in B that is implicitly bound) and used as c (where the argument to c inferred). We will also use fixity and precedence declarations in the style of Twelf to make applications more readable.

Our representation of Mizar in LF is shown in Fig. 2. Note that the encoding presented here is simplified and contains both the ⌜Mizar⌝ and ⌜HIDDEN⌝ signatures. The full encodings are available at [LFE].

It introduces two types *any* and *prop* corresponding to their respective Mizar equivalents and a type *tp* corresponding to Mizar types. The Mizar type *set* which is equivalent with *any*, we encode as having type *tp* and we represent the equivalence through the axiom *set_ax*.

Mizar proofs are encoded as a type family $\vdash$ which is indexed by propositions. Terms p of type $\vdash\ F$ represents proofs of F, and the inhabitation of $\vdash\ F$ represents the provability of F. The argument of $\vdash$ does not need brackets as $\vdash$ has the weakest precedence. Moreover, by convention, the Twelf binders [] and {} always bind as far to the right as is consistent with the placement of brackets.

Higher-order abstract syntax is used to represent binders, e.g., $for\ T\ ([x : any]\, F\ x)$ represents the formula $\forall x : any.x\ is\ T\ implies\ F(x)$. Note that notions belonging exclusively to the pattern level language are defined from constructor level symbols. For example *implies* is defined using *and* and *not* while *ex* is defined from *for* and *not*.

Attributes are encoded using unary predicates which can be thought of as filters for the type argument. Then, they can be clustered by simply taking the conjunction of the applied predicates. Adjectives apply attributes to types giving a new type and corresponding introduction

$$
\begin{array}{lcll}
\%\texttt{sig}\ Mizar\ =\ \{ & & & \\
\quad any & : & \texttt{type} & \\
\quad tp & : & \texttt{type} & \\
\quad prop & : & \texttt{type} & \\
\quad \vdash & : & prop \to \texttt{type} & \%\texttt{prefix}\ 0 \\
\quad is & : & any \to tp \to prop & \%\texttt{infix}\ 30 \\
\quad be & : & any \to tp \to \texttt{type} = [x]\ [t]\ \vdash x\ is\ t & \\
\quad true & : & prop & \\
\quad false & : & prop & \\
\quad not & : & prop \to prop & \%\texttt{prefix}\ 10 \\
\quad and & : & prop \to prop \to prop & \%\texttt{infix}\ 20 \\
\quad implies & : & prop \to prop \to prop = not(A\ and\ (not\ B)) & \%\texttt{infix}\ 25 \\
\quad \vdots & & & \\
\quad for & : & tp \to (any \to prop) \to prop & \\
\quad ex & : & tp \to (any \to prop) \to prop = [t]\,[p]\ not\ (for\ t\ ([x]\ not\ (p\ x))) & \\
\quad \vdots & & & \\
\quad attr & : & tp \to \texttt{type} = [t]\,(any \to prop) & \\
\quad cluster & : & attr\ T \to attr\ T \to attr\ T = [a]\,[b]\,([x]\ \ (a\ x)\ and\ (b\ x)) & \\
\quad adj & : & \{t : tp\}\ attr\ t \to tp & \\
\quad adjI & : & \{x : any\}\ x\ be\ T \to \vdash A\ x \to x\ be\ (adj\ T\ A) & \\
\quad adjE' & : & \{x : any\}\ x\ be\ (adjective\ T\ A) \to \vdash x\ is\ T & \\
\quad adjE' & : & \{x : any\}\ x\ be\ (adjective\ T\ A) \to \vdash A\ x & \\
\quad fraenkel & : & tp \to (any \to prop) \to (any \to any) & \\
\quad \vdots & & & \\
\quad set & : & tp & \\
\quad set\_ax & : & \{x : any\} \vdash x\ is\ set & \\
\quad \vdots & & & \\
\} & & &
\end{array}
$$

Figure 2: LF Signature for Mizar

Parameters		
Name	**Type**	**Comment**
τ	*prop*	the formula

Elaboration	
Name	**Type**
theorem	$\vdash \tau$

Figure 3: Justified Theorem Pattern

and elimination rules ensure the intended meaning is captured by the declarations. Similar introduction/elimination rules exists for the other declarations but are omitted here. The Fraenkel operator (Mizar's version of set comprehension) is encoded to have a type, a unary predicate (the filter) and an unary functor as arguments. Then the type (which in our encoding, as in Mizar, can be thought of as the set of the elements of that type) is filtered with the predicate and functor is mapped to the resulting set. As for *adj*, introduction and elimination rules, which we omit here, ensure the correct meaning.

5.2 Pattern level translation

As discussed in Section 4 OMDoc has three semantic levels, Object, Symbol and Theory levels so we need to "fit" each (processed) Mizar construct into one of these categories:

- Object level contains Mizar terms, types and formulas which are thus represented as OpenMath expressions. Primitive Mizar types are included from the Mizar theory and can be used as OpenMath symbols.
- Symbol level contains Mizar justified theorems, definitions, schemes, notations and registrations. They are represented in OMDoc using *patterns*.
- Theory level is only used for Mizar Articles. Since in Mizar the ontological separation of documents and theories is not explicitly made we represent a Mizar article ϑ as a OMDoc document ϑ containing a single OMDoc theory ϑ which in turn contains the content of the article.

While non-trivial, the object level translation is reasonably straightforward and the few technicalities are discussed in Section 6. The Symbol level is more challenging because many Mizar constructs have no direct equivalent in OMDoc/MMT. So, in order to translate them we use patterns as a meta-language feature which allows for the declaration in OMDoc of Mizar language features. Then, the actual definitions, theorems, schemes, notations or registrations that appear in Mizar articles are represented in OMDoc as *instances* of their respective pattern.

The pattern declarations we used are described below.

5.3 Justified Theorems

In Mizar justified theorems contain a proposition and a proof of that proposition. Since we are not translating proofs (yet) we need only translate the proposition (formula) which results into a MMT Term (object level). So in the OMDoc Mizar article justified theorem is a Pattern that takes a Term (the translation of the Mizar formula) as an argument and expands to a OMDoc constant having a deduction of that proposition as a type and no definition (proof).

5.4 Definitions

There are several types of definitions in Mizar which results to a large number of Patterns in the OMDoc Mizar Article containing the Mizar Primitives.

Parameters		
Name	***Type***	***Comment***
n	int	number of arguments
$\vec{t}$	tp^n	argument types
τ	tp	return type
m	int	number of cases
$\vec{c}$	$(any^n \to prop)^m$	cases
$\vec{r}$	$(any^n \to any)^m$	corresponding results
d	$any^n \to any$	default result

Elaboration	
Name	***Type***
$func$	$\left[any\right]_{i=1}^{n} \to any$
$rtype$	$\Pi_{\vec{x}:any^n} \left[x_i \ is \ t_i\right]_{i=1}^{n} \to func(\vec{x}) \ is \ rt$
$means$	$\Pi_{\vec{x}:any^n} \left[x_i \ is \ t_i\right]_{i=1}^{n} \to$ $\vdash \bigwedge \left[c_i(\vec{x}) \Rightarrow (func(\vec{x}) = r_i(\vec{x}))\right]_{i=1}^{m} \wedge (\bigwedge \left[\neg c(\vec{x})\right]_{i=1}^{m} \Rightarrow (func(\vec{x}) = d(\vec{x})))$

Figure 4: Direct Partial Functor Definition Pattern

Firstly, definitions may be **direct** [4,6,8,9,12,14] or **indirect** [5,7,10,11,13,15] depending whether they use a term or formula to introduce the definiendum. Also, they may be **complete** [6,7,9,11,14,15] or **partial** [4,5,8,10,12,13] depending on whether the cases of the definition span the universe or whether a default case is required. Finally, they may be **functor** [4,5,6,7] **predicate** [8,9], **attribute** [10,11], **mode** [12,13,14,15] or **structure** [16] definitions. See section 3 for Mizar examples of each definition.

Note that there are also **redefinitions** which add a different meaning to a previous extant definition and they are also classified by the above characteristics.

Note that the types (for arguments and the return type) are actually dependently typed, each of them (possibly) depending on any (or all) of the arguments declared before. However, for the sake of simplicity and readability we omit this here and we assume argument types have the type tp rather that $any^k \to tp$. We use $\vec{t}$ as a notation for a sequence and t_i for the i^{th} element of the sequence. Similarly we use $\left[\cdots\right]_{i=k}^{l}$ for iterating through the elements of a sequence from k to l. Since we often have sequences as parameters we consider some operators (e.g. $\to$) to be flexory. Also, we use $\bigvee \vec{t}$ and $\bigwedge \vec{t}$ for the disjunction/conjunction of all elements in a sequence. These are just notational technicalities as in practice any operator with flexory arity can be replaced by a flexory number of fixed arity operators.

Functor Definitions can be both direct and indirect as well as both partial and complete leading to four patterns for functor definitions The parameters are the arguments and return type for the functor itself, the cases and corresponding results and (for partial definitions only) the default result. The elaboration consists of the actual functor (*func*), a statement fixing the proper return type of the functor (*rtype*) and another statement giving the meaning of the functor (*means*). Additionally, for complete definitions, a statement that the cases cover the entire space (and thus a default result is not necessary) is added (*completeness*).

Parameters		
Name	***Type***	***Comment***
n	int	number of arguments
$\vec{t}$	tp^n	argument types
τ	tp	return type
m	int	number of cases
$\vec{c}$	$(any^n \to prop)^m$	cases
$\vec{r}$	$(any^{n+1} \to prop)^m$	corresponding results
d	$any^{n+1} \to prop$	default result

Elaboration	
Name	***Type***
$func$	$\left[set\right]_{i=1}^{n} \to set$
$rtype$	$\Pi_{\vec{x}:any^n} \left[x_i\ is\ t_i\right]_{i=1}^{n} \to func(\vec{x})\ is\ rt$
$means$	$\Pi_{\vec{x}:any^n} \left[x_i\ is\ t_i\right]_{i=1}^{n} \to$
	$\vdash \bigwedge \left[c_i(\vec{x}) \Rightarrow r_i(\vec{x}, func(\vec{x}))\right]_{i=1}^{m} \wedge (\bigwedge \left[\neg c_i(\vec{x})\right]_{i=1}^{m} \Rightarrow d(\vec{x}, func(\vec{x})))$

Figure 5: Indirect Partial Functor Definition Pattern

Parameters		
Name	***Type***	***Comment***
n	int	number of arguments
$\vec{t}$	tp^n	argument types
τ	tp	return type
m	int	number of cases
$\vec{c}$	$(any^n \to prop)^m$	cases
$\vec{r}$	$(any^n \to any)^m$	corresponding results

Elaboration	
Name	***Type***
$func$	$\left[any\right]_{i=1}^{n} \to any$
$rtype$	$\Pi_{\vec{x}:any^n} \left[x_i\ is\ t_i\right]_{i=1}^{n} \to func(\vec{x})\ is\ rt$
$means$	$\Pi_{\vec{x}:any^n} \left[x_i\ is\ t_i\right]_{i=1}^{n} \to \vdash \bigwedge \left[c_i(\vec{x}) \Rightarrow (func(\vec{x}) = r_i(\vec{x}))\right]_{i=1}^{m}$
$completeness$	$\Pi_{\vec{x}:any^n} \left[x_i\ is\ t_i\right]_{i=1}^{n} \to \vdash \bigvee \left[c_i(\vec{x})\right]_{i=1}^{m}$

Figure 6: Direct Complete Functor Definition Pattern

Parameters		
Name	***Type***	***Comment***
n	int	number of arguments
$\vec{t}$	tp^n	argument types
τ	tp	return type
m	int	number of cases
$\vec{c}$	$(any^n \to prop)^m$	cases
$\vec{r}$	$(any^{n+1} \to prop)^m$	corresponding results

Elaboration	
Name	***Type***
$func$	$\left[set\right]_{i=1}^{n} \to set$
$rtype$	$\Pi_{\vec{x}:any^n}\ \left[x_i\ is\ t_i\right]_{i=1}^{n} \to func(\vec{x})\ is\ rt$
$means$	$\Pi_{\vec{x}:any^n} \left[x_i\ is\ t_i\right]_{i=1}^{n} \to \vdash \bigwedge \left[c_i(\vec{x}) \Rightarrow r_i(\vec{x}, func(\vec{x}))\right]_{i=1}^{m}$
$completeness$	$\Pi_{\vec{x}:any^n} \left[x_i\ is\ t_i\right]_{i=1}^{n} \to \vdash \bigvee \left[c_i(\vec{x})\right]_{i=1}^{m}$

Figure 7: Indirect Complete Functor Definition Pattern

Parameters		
Name	***Type***	***Comment***
n	int	number of arguments
$\vec{t}$	tp^n	argument types
m	int	number of cases
$\vec{c}$	$(set^n \to prop)^m$	cases
$\vec{r}$	$(set^n \to prop)^m$	corresponding results
d	$any^n \to prop$	default result

Elaboration	
Name	***Type***
$pred$	$any^n \to prop$
$means$	$\Pi_{\vec{x}:any^n}\ \left[x_j\ is\ t_j\right]_{j=1}^{n} \to$
	$\vdash \bigwedge \left[c_i(\vec{x}) \Rightarrow pred(\vec{x}) \Leftrightarrow r_i(\vec{x})\right]_{i=1}^{m} \wedge (\bigwedge \left[\neg \vec{c_i}(\vec{x})\right]_{i=1}^{m} \Rightarrow (pred(\vec{x}) \Leftrightarrow d(\vec{x})))$

Figure 8: Direct Partial Predicate Definition Pattern

Parameters		
Name	***Type***	***Comment***
n	int	number of arguments
$\vec{t}$	tp^n	argument types
m	int	number of cases
$\vec{c}$	$(set^n \to prop)^m$	cases
$\vec{r}$	$(set^n \to prop)^m$	corresponding results

Elaboration	
Name	***Type***
$pred$	$any^n \to prop$
$means$	$\Pi_{\vec{x}:any^n}\ \left[x_j\ is\ t_j\right]_{j=1}^{n} \to$
	$\vdash \bigwedge \left[c_i(\vec{x}) \Rightarrow pred(\vec{x}) \Leftrightarrow r_i(\vec{x})\right]_{i=1}^{m} \wedge (\bigwedge \left[\neg \vec{c_i}(\vec{x})\right]_{i=1}^{m} \Rightarrow (pred(\vec{x}) \Leftrightarrow d(\vec{x})))$
$completeness$	$\Pi_{\vec{x}:any^n} \left[x_i\ is\ t_i\right]_{i=1}^{n} \to \vdash \bigvee \left[c_i(\vec{x})\right]_{i=1}^{m}$

Figure 9: Direct Complete Predicate Definition Pattern

Predicate Definitions are always direct as they can only be defined using a formula which directly gives the result of the predicate. Thus, there are two patterns, for partial and complete predicate definitions which are given below.

Parameters		
Name	***Type***	***Comment***
n	int	number of arguments
$\vec{t}$	tp^n	definition argument types
τ	$any^n \to tp$	mother type
m	int	number of cases
$\vec{c}$	$(any^n \to prop)^m$	cases
$\vec{r}$	$(any^{n+1} \to prop)^m$	corresponding results
d	$any^{n+1} \to prop$	default result

Elaboration	
Name	***Type***
$attr$	$\Pi_{\vec{x}:any^n}(\tau(\vec{x}) \to prop)$
$mtype$	$\Pi_{\vec{x}:any^n}\Pi_{\vec{p}:\left[x_i\ is\ t_i\right]_{i=1}^n} \to \forall y.y\ is\ (adj\ \vec{\tau}\ (attr\ \vec{x}\ \vec{p})) \Rightarrow y\ is\ \tau(\vec{x})$
$means$	$\Pi_{\vec{x}:any^n}\Pi_{\vec{p}:\left[x_i\ is\ t_i\right]_{i=1}^n} \to \vdash \bigwedge \left[c_i(\vec{x}) \Rightarrow \forall y.(y\ is\ (adj\ \vec{\tau}\ (attr\ \vec{x}\ \vec{p})) \Leftrightarrow r_i(\vec{x},y))\right]_{i=1}^m \wedge$ $(\bigwedge \left[\neg c_i(\vec{x})\right]_{i=1}^m \Rightarrow \forall y.(y\ is\ (adj\ \vec{\tau}\ (attr\ \vec{x}\ \vec{p})) \Leftrightarrow d(\vec{x},y)))$

Figure 10: Indirect Partial Attribute Definition Pattern

Attribute Definitions are always indirect as their meaning is always given as a formula and not by abbreviation. This generates two patterns, for indirect complete and indirect partial attribute definition. Note that the attribute itself only has one argument that is typed by the mother type of the attribute and to which the attribute is applied. However, the mother type itself can be dependently typed (e.g. Element of X) so the arguments are exclusively used for the mother type.

Mode Definitions may be direct or indirect, complete or partial so there are four patterns. Modes define new types by applying attributes (or clusters of attributes) to an existing type (direct) or by giving the meaning as a formula (indirect). The patterns are presented below.

Structure Definitions in Mizar's pattern language act as syntactic sugar since internally they expand to several constructors. Internally, for each structure definitions, the Mizar processor generates a structure type, a structure constructor and a finite sequence of selectors, one for each structure field. Since the constructor number is variable (depending on the number of fields) we represent structures by a family of patterns, with each having a fixed number of fields. Then a structure definition becomes an instance of the corresponding pattern, depending its number of fields. Also, note that in the structure pattern presented in Fig. 16, *struct*, *aggr* and *sel* also have definitions based on list constructors and selectors defined in the Mizar LF article. We omit these here but we mention that Mizar selectors act as projection operations over structures and the natural projection properties needed for adequacy ($\pi_i(\left[x_1, \cdots, x_n\right]) = x_i$ and $\left[\pi_1(s), \cdots, \pi_n(s)\right] = s$) are inherited from those of lists.

5.5 Schemes

Scheme patterns have as parameters a list of arguments, a list of premises and a proposition. Then, the elaboration states that given the premises the scheme proposition holds.

5.6 Notation

There are two patterns for Notations, one for Antonymic Notation and one of Synonymic notation.

5.7 Registrations

One can create new types by applying attributes (or clusters of attributes) to an existing type. However, in Mizar, types must be non-empty so before attributes can be used a proof of non-

Parameters		
Name	***Type***	***Comment***
n	int	number of arguments
$\vec{t}$	tp^n	definition argument types
τ	$any^n \to tp$	mother type
m	int	number of cases
$\vec{c}$	$(any^n \to prop)^m$	cases
$\vec{r}$	$(any^{n+1} \to prop)^m$	corresponding results

Elaboration	
Name	***Type***
$attr$	$\Pi_{\vec{x}:any^n}(\tau(\vec{x}) \to prop)$
$mtype$	$\Pi_{\vec{x}:any^n}\Pi_{\vec{p}:\left[x_i\ is\ t_i\right]_{i=1}^n} \to \forall y.y\ is\ (adj\ \vec{\tau}\ (attr\ \vec{x}\ \vec{p})) \Rightarrow y\ is\ \tau(\vec{x})$
$means$	$\Pi_{\vec{x}:any^n}\Pi_{\vec{p}:\left[x_i\ is\ t_i\right]_{i=1}^n} \to \vdash \bigwedge \left[c_i(\vec{x}) \Rightarrow \forall y.(y\ is\ (adj\ \vec{\tau}\ (attr\ \vec{x}\ \vec{p})) \Leftrightarrow r_i(\vec{x},y))\right]_{i=1}^m$
$completeness$	$\Pi_{\vec{x}:any^n}\left[x_i\ is\ t_i\right]_{i=1}^n \to \vdash \bigvee \left[c_i(\vec{x})\right]_{i=1}^m$

Figure 11: Indirect Complete Attribute Definition Pattern

Parameters		
Name	***Type***	***Comment***
n	int	number of arguments
$\vec{t}$	tp^n	argument types
m	int	number of cases
$\vec{c}$	$(set^n \to prop)^m$	cases
$\vec{r}$	$(set^n \to tp)^m$	corresponding results
d	$any^n \to tp$	default result

Elaboration	
Name	***Type***
$mode$	$any^n \to tp$
$means$	$\Pi_{\vec{x}:any^n}\ \left[x_j\ is\ t_j\right]_{j=1}^n \to$ $\vdash \bigwedge \left[c_i(\vec{x}) \Rightarrow mode(\vec{x}) = r_i(\vec{x})\right]_{i=1}^m \wedge (\bigwedge \left[\neg c_i(\vec{x})\right]_{i=1}^m \Rightarrow (mode(\vec{x}) = d(\vec{x})))$

Figure 12: Direct Partial Mode Definition Pattern

Parameters		
Name	***Type***	***Comment***
n	int	number of arguments
$\vec{t}$	tp^n	argument types
m	int	number of cases
$\vec{c}$	$(set^n \to prop)^m$	cases
$\vec{r}$	$(set^{n+1} \to prop)^m$	corresponding results
d	$any^{n+1} \to prop$	default result

Elaboration	
Name	***Type***
$mode$	$any^n \to tp$
$means$	$\Pi_{\vec{x}:any^n}\ \left[x_j\ is\ t_j\right]_{j=1}^n \to$ $\vdash \bigwedge \left[c_i(\vec{x}) \Rightarrow r_i(\vec{x}, mode(\vec{x}))\right]_{i=1}^m \wedge (\bigwedge \left[\neg c_i(\vec{x})\right]_{i=1}^m \Rightarrow d(\vec{x}, mode(\vec{x})))$

Figure 13: Indirect Partial Mode Definition Pattern

Parameters		
Name	***Type***	***Comment***
n	int	number of arguments
$\vec{t}$	tp^n	argument types
m	int	number of cases
$\vec{c}$	$(set^n \to prop)^m$	cases
$\vec{r}$	$(set^n \to tp)^m$	corresponding results

Elaboration	
Name	***Type***
$mode$	$any^n \to tp$
$means$	$\Pi_{\vec{x}:any^n} \left[x_j \ is \ t_j\right]_{j=1}^{n} \to \vdash \bigwedge \left[c_i(\vec{x}) \Rightarrow mode(\vec{x}) = r_i(\vec{x})\right]_{i=1}^{m}$
$completeness$	$\Pi_{\vec{x}:any^n} \left[x_i \ is \ t_i\right]_{i=1}^{n} \to \vdash \bigvee \left[c_i(\vec{x})\right]_{i=1}^{m}$

Figure 14: Direct Complete Mode Definition Pattern

Parameters		
Name	***Type***	***Comment***
n	int	number of arguments
$\vec{t}$	tp^n	argument types
m	int	number of cases
$\vec{c}$	$(set^n \to prop)^m$	cases
$\vec{r}$	$(set^{n+1} \to prop)^m$	corresponding results

Elaboration	
Name	***Type***
$mode$	$any^n \to tp$
$means$	$\Pi_{\vec{x}:any^n} \left[x_j \ is \ t_j\right]_{j=1}^{n} \to \vdash \bigwedge \left[c_i(\vec{x}) \Rightarrow r_i(\vec{x}, mode(\vec{x}))\right]_{i=1}^{m}$
$completeness$	$\Pi_{\vec{x}:any^n} \left[x_i \ is \ t_i\right]_{i=1}^{n} \to \vdash \bigvee \left[c_i(\vec{x})\right]_{i=1}^{m}$

Figure 15: Indirect Complete Mode Definition Pattern

Parameters		
Name	***Type***	***Comment***
n	int	number of arguments
$\vec{t}$	tp^n	argument types
$\tau 1$	$any^n \to tp$	type of field 1
$\vdots$	$\vdots$	$\vdots$
τm	$any^n \to tp$	type of field m

Elaboration	
Name	***Type***
$struct$	$any^n \to tp$
$aggr$	$\Pi_{\vec{x}:any^n}\left[x_i \ is \ t_i\right]_{i=1}^{n} \to \Pi_{f1:any} \ (f1 \ is \ \tau 1) \to \cdots \to \Pi_{fm:any} \ (fm \ is \ \tau m) \to any$
$aggr\_prop$	$\Pi_{\vec{x}:any^n}\Pi_{\vec{p}:\left[x_i \ is \ t_i\right]_{i=1}^{n}} \ \Pi_{f1:any}\Pi_{\rho 1:(f1 \ is \ \tau 1)} \ \cdots \ \Pi_{fm:any}\Pi_{\rho m:(fm \ is \ \tau m)}$ $\to (aggr \ \vec{x} \ \vec{p} \ f1 \ \rho 1 \ \cdots \ fm \ \rho m) \ is \ struct$
$\upsilon 1$	$\Pi_{\vec{x}:any^n}\Pi_{\vec{p}:\left[x_i \ is \ t_i\right]_{i=1}^{n}} \to struct(\vec{x}) \to any$
$\upsilon 1\_prop$	$\Pi_{\vec{x}:any^n}\Pi_{\vec{p}:\left[x_i \ is \ t_i\right]_{i=1}^{n}} \ \Pi_{s:struct(\vec{x})} \to \upsilon 1(\vec{x},\vec{p},s) \ is \ \tau 1$
$\vdots$	$\vdots$
υm	$\Pi_{\vec{x}:any^n}\Pi_{\vec{p}:\left[x_i \ is \ t_i\right]_{i=1}^{n}} \to struct(\vec{x}) \to any$
υm_prop	$\Pi_{\vec{x}:any^n}\Pi_{\vec{p}:\left[x_i \ is \ t_i\right]_{i=1}^{n}} \ \Pi_{s:struct(\vec{x})} \to \upsilon 1(\vec{x},\vec{p},s) \ is \ \tau m$

where m is the number of fields

Figure 16: Structure Definition Pattern

Parameters		
Name	***Type***	***Comment***
n	int	number of arguments
$\vec{t}$	tp^n	argument types
m	int	number of assumptions
$\vec{a}$	$(set^n \to prop)^m$	assumptions
p	$set^n \to prop$	scheme proposition

Elaboration	
Name	***Type***
$scheme$	$\Pi_{\vec{x}:any^n}\left[x_i \ is \ t_i\right]_{i=1}^{n} \to \vdash \bigwedge \left[a_i(\vec{x})\right]_{i=1}^{m} \to \vdash p(\vec{x})$

Figure 17: Scheme Pattern

Parameters		
Name	**Type**	**Description**
n	int	number of arguments
$\vec{t}$	tp^n	argument types
ν	$any^n \to prop$	the construct being denoted

Elaboration		
Name	**Type**	**Definition**
$notation$	$\Pi_{\vec{x}:any^n} \ \left[x_i \ is \ t_i\right]_{i=1}^{n} \to prop$	$\Lambda_{\vec{x}} \ \Lambda_{\vec{p}} \ \nu(\vec{x})$

Figure 18: The Synonymic Notation Pattern

Parameters		
Name	**Type**	**Description**
n	int	number of arguments
$\vec{t}$	tp^n	argument types
ν	$any^n \to prop$	the construct being denoted
Elaboration		
Name	**Type**	**Definition**
$notation$	$\Pi_{\vec{x}:any^n} \left[x_i \text{ is } t_i\right]_{i=1}^n \to prop$	$\Lambda_{\vec{x}}\, \Lambda_{\vec{p}}\, \neg\nu(\vec{x})$

Figure 19: The Antonymic Notation Pattern

Parameters		
Name	**Type**	**Description**
$\vec{n}$	int	number of arguments
$\vec{t}$	tp^n	argument types
τ	$any^n \to tp$	applicable type
m	int	number of attributes
$\vec{a}$	$any^{n+1} \to prop^m$	attributes to be clustered
Elaboration		
Name	**Type**	
$registration$	$\Pi_{\vec{x}:any^n} \left[x_i \text{ is } t_i\right]_{i=1}^n \vdash\ \exists x.x \text{ is } \left(\left[a_i(\vec{x})\right]_{i=1}^n \ @\ \tau\right)$	

Figure 20: The Registration Pattern

emptiness is required. Similarly, when one clusters several attributes, one must prove that their intersection is non-empty in order to *register* that cluster of attributes as valid. We use the (flexory) notation @ for applying attributes to a type.

6 Translation

The source files for the translation are (some of) the output files of the Mizar processor after some XSLT post-processing [Urb05].

The translation begins by loading the article imports from the `.sgl` file, then the notational and presentation information are read from the `.idx`, `.dcx` and `.frx` files and stored in a dictionary.. Finally, the content file (with the constructor language representation of the article) is parsed and translated, with notations and presentation information being looked up in the dictionary whenever necessary.

Note that we use the absolute XML files where inter-article dependencies are resolved and

File	Comment
`.xml`	contains the constructor language version of the article (encoded as xml)
`.absxml`	same as .xml but with absolute paths resolved by XSLT post-processing
`.sgl`	contains the names of the articles imported by the current article
`.idx`	contains names for the variables that occur in the article
`.dcx`	contains symbol names (for functors, predicates, etc..)
`.frx`	contains extra presentational information such as the number of left and right arguments for functors or predicates

Figure 21: Source Files Used

marked. For example, the description of the basic Mizar type *set* as generated by the system is:

```
<Typ kind="M" nr="1">
  <Cluster/>
</Typ>
```

Then, the XSLT processing we have:

```
<Typ kind="M" nr="1" aid="HIDDEN" absnr="1">
  <Cluster/>
</Typ>
```

so we know it refers to the first mode definition in article HIDDEN.

The translation process consists of two steps, the parser and the translator:

- The Mizar **parser** is responsible for organizing the output of the Mizar compiler (together with the output of the XSLT Post-processor) into coherent data structures that correspond as much as possible to the Mizar pattern-level language. During relevant information is extracted from the processed Mizar files and stored into Scala classes. Because the internal structure of Mizar is much different from the of OMDoc the extracting of information is sometimes tricky, as information the correspond to one "entity" in the Mizar pattern-level language are sometimes stored in different files or different parts of the same file. The main task of the parser is to gather this information and construct a level where as much of the pattern-level language as possible is recovered. The parser contains a *Parsing Controller* which acts as a state machine and directs the parsing. The parsing controller first loads notation and presentation information from auxiliary Mizar output files and stores them in a dictionary which is then used for lookup during the translation. Following the processing done by the parser we get a Mizar Article containing a List of Mizar Elements, which may be Definitions, Registrations, Notations, Schemes or Justified Theorems.

- Then, the **translator**, in turn, processes the Mizar Article and creates an MMT Article by translating each element (and if necessary recursing into it). As described in section 5 each top level Mizar element is translated into an instance (or more for structures) while Mizar propositions, terms and types are translated into MMT terms. In the translation step, the Mizar Scala classes are processed into MMT classes so the the MMT API can then be used to generate and/or manipulate the OMDoc files. The *Translation Controller* directs the flow of the translator and is responsible for looking up names and notations as well as keeping track of variable scopes. This is necessary because Mizar uses de Bruijn indexes internally for bound variables while OMDoc uses named variables and the translator must be aware of scopes to translate adequately. The translation step must be made as transparently as possible so to preserve adequacy. The MMT Checker is able to check the validity of MMT/OMDoc documents but one must further ensure that the content of the translated documents is preserved and that the result library encodes the same mathematical information as the MML.

The architecture of the system is described if Fig. 22.

Since there are many primitives in Mizar the packages are split into files which represent particular parts of the Mizar schema. These parts are split in such a way as to be consistent throughout the translation so that each package has the same distribution.

- *Article* contains the **Article** and the top level classes (**DefinitionBlock**, **JustifiedTheorem**, **NotationBlock**, etc.)

- *Definition* handles the processing of **Definition**s, regardless of kind .

- *Proposition* includes **Proposition** and **Formula** and thus will deal with parsing and translating formulas

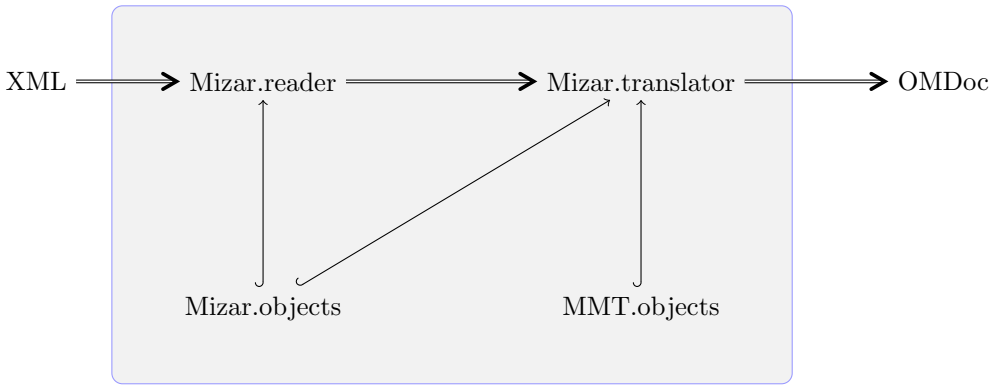

Figure 22: System Architecture

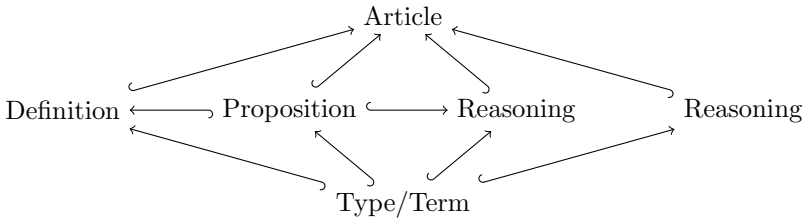

Figure 23: Data Structures Architecture

- *Type/Term* handles parsing **Type**s and **Term**s and thus also deals with resolving variables with respect to binders
- *Reasoning* deals with the assumptions needed by the definitions
- *Scheme/Reg* covers **Scheme**s and **Registration**s

The dependencies of these elements are described in Fig. 23

Overall, translator consists of 88 Scala classes and 47 Scala objects, which are available at `https://svn.kwarc.info/repos/MMT/src/mmt-mizar`.

The entire workflow for generating the OMDoc files consists of 4 steps.

1. Mizar is run over the whole library transforming `miz` input files to `xml`. This represents the state before the work presented here.
2. Our translator reads all `xml` files (in dependency order) and parses them into custom Scala classes that model the constructor level Mizar language.
3. The translator then translates each article into Scala classes for OMDoc. The latter are part of the MMT tool, which serializes them as `omdoc` files. At this point the Mizar-specific part of the translation is over.
4. The generic algorithms in the MMT tool elaborate the pragmatic OMDoc to core OMDoc.

Running the steps 2-4 of the pipeline on all 1129 files of the MML takes 41:40 minutes using a Intel Core i5-2410M Processor and 4GB of RAM. A fine grained breakdown is given in the table below.

Format	Total file size (raw/zipped)	Generation time
Mizar source	77.5 MB / 13.5 MB	—
Mizar XML	8.6 GB / 425.0 MB	—
pragmatic OMDoc	894.8 MB / 22.7 MB	29:46 min
core OMDoc	1015.3 MB / 25.3 MB	11:54 min

The resulting files are available at https://tntbase.mathweb.org/repos/oaff/mml/

7 Conclusion

In this report we have described a translation of the Mizar mathematical language to the OMDoc format. At the same time, this may be considered as one step towards translating all the major libraries of formalized mathematics into one format so that we can achieve seamless integration between all such libraries.

Currently, the Mizar to OMDoc translation does not cover proofs in the Mizar library (arguably one of the most important parts), and we are planning to extend the coverage soon.[4] To the extent that Mizar can export proofs, this will be straightforward using our existing formalization of Mizar's inference rules in LF. The OMizar format currently only exists at the data structure level in the MMT system. We also want to give it an XML syntax, so that it can be used as a more accessible representation format for Mizar– and thus interoperability and/or storage format. We hope that this will also inform us in the endeavor of creating a truly universal pragmatic level of the OMDoc format.

References

[Ban03] Grzegorz Bancerek. On the structure of Mizar types. *Electronic Notes in Theoretical Computer Science*, 85(7), 2003.

[Ban06a] Grzegorz Bancerek. Automatic translation in Formalized Mathematics. *Mechanized Mathematics and Its Applications*, 5(2):19–31, 2006.

[Ban06b] Grzegorz Bancerek. Information retrieval and rendering with MML Query. In Jon Borwein and William M. Farmer, editors, *Mathematical Knowledge Management (MKM)*, number 4108 in LNAI, pages 266–279. Springer Verlag, 2006.

[BK07] Grzegorz Bancerek and Michael Kohlhase. Towards a Mizar Mathematical Library in OMDoc format. In R. Matuszewski and A. Zalewska, editors, *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, volume 10:23 of *Studies in Logic, Grammar and Rhetoric*, pages 265–275. University of Białystok, 2007.

[DW97] Ingo Dahn and Christoph Wernhard. First order proof problems extracted from an article in the Mizar Mathematical Library. In Ulrich Furbach and Maria Paola Bonacina, editors, *Proceedings of the International Workshop on First order Theorem Proving*, number 97-50 in RISC-Linz Report Series, pages 58–62. Johannes Kepler Universität Linz, 1997.

[HHP93] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.

[HST94] R. Harper, D. Sannella, and A. Tarlecki. Structured presentations and logic representations. *Annals of Pure and Applied Logic*, 67:113–160, 1994.

[IR11] M. Iancu and F. Rabe. Formalizing Foundations of Mathematics. *Mathematical Structures in Computer Science*, 21(4):883–911, 2011.

[JFM] Journal of formalized mathematics. http://www.mizar.org/JFM.

[Koh06] Michael Kohlhase. OMDoc – *An open markup format for mathematical documents [Version 1.2]*. Number 4180 in LNAI. Springer Verlag, August 2006.

[4] Note that handling of proofs has been typically delayed to the second phase in similar Mizar exporting projects like MPTP and MML Query, because a lot of useful functionality can be developed already without proofs.

[KRZ10] Michael Kohlhase, Florian Rabe, and Vyacheslav Zholudev. Towards MKM in the large: Modular representation and scalable software architecture. In Serge Autexier, Jacques Calmet, David Delahaye, Patrick D. F. Ion, Laurence Rideau, Renaud Rioboo, and Alan P. Sexton, editors, *Intelligent Computer Mathematics*, number 6167 in LNAI. Springer Verlag, 2010.

[LFE] LF Encodings. Available at: `http://alpha.tntbase.mathweb.org/repos/cds/source/foundations/mizar`.

[PS99] F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. *Lecture Notes in Computer Science*, 1632:202–206, 1999.

[RK08] F. Rabe and M. Kohlhase. An Exchange Format for Modular Knowledge. In G. Sutcliffe, P. Rudnicki, R. Schmidt, B. Konev, and S. Schulz, editors, *Proceedings of the LPAR Workshops on Knowledge Exchange: Automated Provers and Proof Assistants, and The 7th International Workshop on the Implementation of Logics*, volume 418 of *CEUR Workshop Proceedings*, pages 50–68. CEUR-WS.org, 2008.

[RK11] Florian Rabe and Michael Kohlhase. A scalable module system. Manuscript, submitted to Information & Computation, 2011.

[RS09] F. Rabe and C. Schürmann. A Practical Module System for LF. In J. Cheney and A. Felty, editors, *Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP)*, volume LFMTP'09 of *ACM International Conference Proceeding Series*, pages 40–48. ACM Press, 2009.

[TB85] A. Trybulec and H. Blair. Computer Assisted Reasoning with MIZAR. In A. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28, 1985.

[TR99] Andrzej Trybulec and Piotr Rudnicki. On equivalents of well-foundedness. *Journal of Automated Reasoning*, 23(3-4):197–234, 1999.

[Try90] Andrzej Trybulec. Tarski Grothendieck set theory. *Formalized Mathematics*, 1(**1**):9–11, 1990.

[Urb05] Josef Urban. XML-izing Mizar: making semantic processing and presentation of MML easy. submitted to MKM 2005, available online at http://ktiml.mff.cuni.cz/∼urban/mizxml.ps, 2005.

[Urb06a] Josef Urban. MPTP 0.2: Design, implementation, and initial experiments. *J. Autom. Reasoning*, 37(1-2):21–43, 2006.

[Urb06b] Josef Urban. XML-izing Mizar: making semantic processing and presentation of MML easy. In Michael Kohlhase, editor, *Mathematical Knowledge Management, MKM'05*, number 3863 in LNAI, pages 346 – 360. Springer Verlag, 2006.

[US08] Josef Urban and Geoff Sutcliffe. ATP-based cross-verification of Mizar proofs: Method, systems, and first experiments. *Mathematics in Computer Science*, 2(2):231–251, 2008.

[Wie99] Freek Wiedijk. Mizar: An impression, 1999.