Mathematical Knowledge Management in Mizar

Piotr Rudnicki
Dept. of Computing Science, University of Alberta, Edmonton, Canada
piotr@cs.ualberta.ca

Andrzej Trybulec
Institute of Informatics, University in Białystok, Białystok, Poland
trybulec@math.uwb.edu.pl

September 3, 2001

Abstract

We report on how mathematics is done in the Mizar system. Mizar offers a language for writing mathematics and provides software for proof-checking. Mizar is used to build Mizar Mathematical Library (MML). This is a long term project aiming at building a comprehensive library of mathematical knowledge. The language and the checking software evolve and the evolution is driven by the growing MML.

1 Introduction

The Mizar language is the language of practical formalization of mathematics developed by A. Trybulec. The main goal for its original design and further development was a formal system close to the mathematical jargon used in publications but at the same time simple enough to enable computerized processing, in particular mechanical verification of correctness.

For the above reasons, the logical basis of Mizar is formed by:

- the system of natural deduction by S. Jaśkowski [13] (see also [18]), which is considered an adequate reconstruction of proof styles used in mathematical publications. Similar systems were independently described by K. Ono [20] and F. B. Fitch [9].

- the notion of an obvious inference instead of using some fixed set of inference rules. This notion has been addressed in the past by M. Davis [7], in the Kiev project [8] and recently by H. Friedman [10].

At the beginning of the project, there were plans to test the Mizar language and software in building libraries based on various axiomatics: ZFC for 'normal' mathematics, Peano axioms for theoretical arithmetic, etc. Because of substantial effort needed do build such libraries, a decision was made to focus on developing one such library (MML) based on the Tarski-Grothendieck set theory. In the 1980's, some experiments were carried out with the Morse-Kelley theory of classes but they have been discontinued for unresolved problems in mechanical processing of relationships between sets and classes.

We would like to stress the distinction between MIZAR as a formal system of general applicability (which as such has little in common with any set theory) and the specific application of MIZAR in developing MML which is entirely based on a set theory (see Appendix A). In building MML, the MIZAR language and set theory provide an environment in which all further mathematics is developed. This development is *definitional*: new mathematical objects can be defined provided we supply a model for them in the already available theories.

Currently, the development of MML(see `http://mizar.org`) is the main activity in the MIZAR project as it is believed that only substantial experience may help in improving the system. There has been a focused effort to advance the contents of MML in some areas:

- the theory of continuous lattices through translation of the entire main text of [11]. This theory involves diverse mathematics and also is concerned with quite recent developments. The articles in the series `WAYBEL` cover this material with the necessary background material put into articles in the series named `YELLOW`;
- the proof of the Jordan curve theorem is being continued, articles in the series `GOBOARD` and `JORDAN`;
- reasoning about computations, articles in the series `AMI` and `SCM`;
- algebra towards symbolic computation by following [4].

The management of mathematical knowledge in MIZAR can be considered at different levels.

- When writing a single proof we have to deal with proof structuring, forming auxiliary lemmas, referencing available knowledge inside and outside the proof and forming individual inference steps.
- When writing a MIZAR article (containing usually a number of theorems and definitions which require proofs) we first establish a local conceptual basis for our work by importing material from MML. An article is usually dealing with a restricted area of knowledge and the number of notions that are needed is typically small. Since processing speed depends on the size of the local environment it is important that the environment is reasonably small. While this may not seem as a big problem now, we anticipate it becoming serious when MML grows.
- Once an article is finished, it is included into MML and thus made available for referencing from other articles. As of August 2001, MML contains 700 articles which include some 32000 mathematical facts and some 6000 definitions. MML undergoes revisions as a result of the evolution of the MIZAR language and software. An article is also translated into several other representations which are available on the WWW.

Below, we present some issues of knowledge management at these three levels.

2 An article and its processing

A MIZAR article is written as a text file and consists of two parts: the *Environment Declaration* and the *Text Proper* (see Figure 1). The *Environment Declaration* begins with `environ` and consists of *Directives*. The *Text Proper* is a sequence of *Sections*, each starting with `begin` and consisting of a sequence of *Text Items*. The division of the *Text Proper* into sections is only for editing purposes and has no impact on the correctness of an article. This division is used for sections when typesetting MIZAR abstracts in TEX for publication in *Formalized Mathematics*.

```
environ
        Environment-Declaration
begin
        Text-Proper
```

Figure 1: The overall structure of a MIZAR article.

The two parts of an *Article* are processed by two separate programs: the ACCOMMODATOR and the VERIFIER. The ACCOMMODATOR processes the *Environment Declaration* and creates the local *Environment*, which consists of a number of working files in which the information requested in *Directives* and imported from the data base is stored.

The VERIFIER has no direct communication with the data base and checks the correctness of the *Text Proper* using only the information stored in the local *Environment* files. The VERIFIER consists of three phases the PARSER, the ANALYZER and the CHECKER. The efficient mechanism for importing the information from the MML into the local *Environment* files is of utmost importance and its design presents a substantial challenge; the ACCOMMODATOR evolves probably faster than other components of MIZAR.

2.1 Some remarks about the language

The MIZAR language includes the standard set of first order logical connectives for forming formulae and syntactic constructs for writing proofs. The language provides means for using free second order variables in forming schemes of formulae (infinite families of formulae). An introductory information about MIZAR with numerous examples can be found in [22, 25, 19].

MIZAR offers a number of definitional facilities. Each definition defines a new constructor later used in syntactic constructions, and gives its syntax and meaning.

In MIZAR terminology, predicates are constructors of atomic formulae, modes are constructors of types, functors are constructors of terms, and attributes are constructors of adjectives. The syntactic *format* of a constructor specifies the symbol of the constructor and the place and number of arguments. The format of a constructor together with the information about the types of arguments is called a *pattern*. The formats are used for parsing and the patterns for identifying constructors. A constructor may be represented by different patterns as synonyms and antonyms are allowed.

The definitions of functors, modes and clusters of adjectives require correctness conditions: for a new functor we have to demonstrate that it uniquely denotes an object, for a mode that it constructs non empty types, for a cluster of adjectives we have to show that there are objects with all the corresponding properties.

We borrowed the name "functor" from [21], p. 148:

> ... some signs in the formalized language should correspond to the mappings and

> functions being examined. These signs are called *functors*, or—more precisely—*m-argument functors* provided they correspond to m-argument mappings from objects to objects ($m = 1, 2, ...$).

Mizar functors must not be confused with functors as used in category theory.

2.2 The Environment directives

There are three kinds of *Directives*: *Vocabulary Directives*, *Library Directives* and *Requirements Directives*.

A *Vocabulary* is a text file (in extended ASCII) in which symbols are defined. The symbols are qualified with their kind (predicate, functor, mode, structure, selector, attribute, left or right functor bracket) and are used for lexical analysis. A *Vocabulary Directive* has the form

`vocabulary` *Vocabulary-Name, ..., Vocabulary-Name* ;

and requests that all symbols from the listed vocabularies be included in the local environment. Vocabularies are independent of Mizar articles.

Library Directives request information from the data base for inclusion in the local environment used later by the Verifier to check the article.

The conceptual framework of an article is imported by the directive:

`constructors` *Article-Name, ..., Article-Name* ;

The notation used for the concepts come through the directive:

`notation` *Article-Name, ..., Article-Name* ;

The library directives work at the level of Mizar articles and for instance, including notation from an article includes all notation defined in an article.

The `constructors` directive first imports all constructors defined in the listed articles and then recursively all other constructors needed to understand them. As a result, if a local environment contains a constructor then it also contains the constructors occurring in types of its arguments and in case of functors also in the result type.

The remaining directives are:

- `clusters` *Article-Name, ..., Article-Name* ;

  imports, from the listed articles, the registrations of clusters. There is a number of different types of clusters. Each registered cluster needs to satisfy certain correctness conditions. Cluster registrations state relationships among adjectives, modes, and functors, and are automatically processed by the Analyzer and the Checker.

- `definitions` *Article-Name, ..., Article-Name* ;

  requests definientia that can be used in proving by definitional expansion without mentioning the definition's name.

- `theorems` *Article-Name, ..., Article-Name* ;

  enables referring to theorems and definitional theorems from the listed articles.

- `schemes` *Article-Name, ..., Article-Name* ;

  gives access to schemes, which are theorems with second order free variables.

- `requirements` *Req-Name, ..., Req-Name* ;

  gives access to built-in features; there are only four such requirements groups at the moment named `ARYTM`, `BOOLE`, `SUBSET` and `REAL`.

As mentioned before, the *Environment Declaration* is processed by the program called ACCOMMODATOR. The basic directive is `constructors` which imports constructors from the listed articles and sets the conceptual framework of the local environment. The directives `clusters`, `definitions`, `theorems` and `schemes` then import respective library items from the listed articles provided the constructors needed to understand the items already have been imported. This approach aims at controlling the local environment to be small (see Section 6). It seems, that with the growing size of MML there is a need for such control.

2.3 The Text Proper

Each section of *Text Proper* is a sequence of *Text Items*. There are the following kinds of *Text Items*:

- a *Reservation* is used to reserve identifiers for a type. If a variable has an identifier reserved for a type, and no explicit type is stated for the variable, then its type defaults to the type for which its identifier was reserved.

- a *Definition Block* contains a sequence of *Definition Items*, each defining or redefining a constructor or a cluster. Each definition and redefinition of a constructor requires a justification of its correctness. For instance, when defining a functor, one has to justify conditions of its existence and uniqueness; when redefining a functor, one has to demonstrate that the redefinition is coherent with respect to the original definition of the functor.

- a *Structure Definition* introduces a new structure which is an entity that consists of a number of fields (members) that are accessible by selectors.

- a *Theorem* announces a proposition that is put into the data base and can then be referenced from other articles.

- a *Scheme* announces a scheme which is a theorem with second order free variables and is accessible from other articles. Schemes can be perceived as inference rules and they have to be proven before being used. The replacement axiom is formulated as a scheme named `Fraenkel`, see Appendix A.

- *Auxiliary Items* form those parts of a MIZAR article that are local to the article and are not exported to the data base files (e.g. auxiliary lemmas, definitions of local predicates, local functions, and variables).

Most *Text Items* require a *Justification*, which can be either a *Straightforward Justification*, a *Proof*, or a *Scheme Justification*. MIZAR permits a multitude of proof structures in the spirit of natural deduction—too many to discuss them here, see [19], pp. 8–22.

The *Straightforward Justification* takes the form:

`by` *Reference, ..., Reference* `;`

where a *Reference* is either a *Private Reference*—a reference to a statement in the current article, or a *Library Reference*—a reference to a theorem stored in the data base. The latter has two forms:

Article-Name `:` *Theorem-Number*

or

Article-Name `:` `def` *Definition-Number*

where the latter refers to the so called definitional theorem, a theorem automatically created from a definition.

The *Scheme Justification* is of the form:

`from` *Scheme-Name* `(` *Reference, ..., Reference* `)`

Scheme-Names are global and are not prefixed with the name of the article where they were introduced. The *References* give the premises of the scheme.

2.4 Processing

The ACCOMMODATOR creates the local *Environment* files for an article based on the *Environment Directives*. Then the VERIFIER works with this local environment never contacting the data base. The VERIFIER work is split into three phases:

- The PARSER is a relatively complicated program because of the rich MIZAR syntax, multi-way overloading of names, and the presence of newly defined formats of constructors and their precedences.

- The ANALYZER identifies constructors based on the available patterns and this involves processing type information. The ANALYZER also checks the correctness of proof structures, computes the formulae demonstrated in *diffuse* statements, and processes clusters of attributes.

- The CHECKER checks the straightforward justifications by treating each as an inference of the form

$$\frac{premise_0,\ premise_1,\ \ldots,\ premise_k}{conclusion}$$

  which is transformed into

$$\frac{premise_0\ \&\ premise_1\ \&\ \ldots\ \&\ premise_k\ \&\ \texttt{not}\ conclusion}{\texttt{contradiction}}$$

  If the checker finds the conjunction contradictory then the original inference step is accepted, see Section 3. The checker may not accept an inference that is logically correct; to get the inference accepted one has to split it into a sequence of 'smaller' ones, or possibly use a proof structure.

- The *Schemetizer* checks the correctness of *Scheme Justifications* by pattern matching the premises and the conclusion of the scheme definition with the actual premises and the actual proposition being justified.

3 Obvious inferences

One may write *Straightforward Justifications* at a very low level, using a single basic inference rule at an inference step. Let the basic inference rule be one of the introduction/elimination rules of natural deduction. All correct applications of single basic rules will be accepted by the checker; however, the rule involved in an inference step cannot be named as MIZAR does not provide syntax for such naming. (One can put the name of the inference rule in a comment; this is how MIZAR is sometimes used in teaching introductory logic).

Using only a single basic inference rule at an inference step would certainly make even a simple proof very long. Therefore, the MIZAR checker when verifying an inference step uses a more complicated decision procedure which recognizes a set of 'obvious' inferences. This decision procedure stresses the processing speed, not power. The characterization of 'obvious' inferences is best given by describing the checker actions. At this moment, we cannot offer a high-level explanation of what constitutes an 'obvious' inference. In the description below (based on a write-up by F. Wiedijk) we only give a general idea of the checker's work; many details are not mentioned.

Given a conjunction to be refuted:

$$premise_0 \mathbin{\&} premise_1 \mathbin{\&} \ldots \mathbin{\&} premise_k \mathbin{\&} \texttt{not}\ conclusion$$

the checker works in three stages: PRECHECKER, EQUALIZER and UNIFIER.

Prechecker

All formulae are converted into a normal form:

- Only the following logical connectives are used: &, `not`, and `for` (the universal quantifier). All other connectives are eliminated using the well known equivalences.
- Conjunctions are represented as lists of conjuncts. An empty list is a unity of conjunction and serves as the logical value *true*. A negated empty list represents a `contradiction`.
- Double negations are eliminated.
- The distributivity of the universal quantifier over conjunction and the distributivity of the existential quantifier over disjunction are used to 'push down' the quantifiers. Fictitious quantifiers are removed.
- Disjunctive normal form of the starting conjunction is created resulting in:

  $$(q_{0,0} \mathbin{\&} \ldots \mathbin{\&} q_{0,k_0})\ \texttt{or}\ \ldots\ \texttt{or}\ (q_{n,0} \mathbin{\&} \ldots \mathbin{\&} q_{n,k_n})$$

  A disjunct is stored as a collection of formulae with no repetitions and at this level conjunction is treated as a commutative connective. Contradictory disjuncts are not considered any further. If the original inference is a propositional tautology then the disjunctive normal form is empty and the inference is accepted.

- If a disjunct contains an existential sentence then PRECHECKER applies the existential elimination rule to this sentence by introducing a new constant and replacing the sentence after appropriate substitution. A new disjunctive form for this disjunct is built.

After the above steps, we have a collection of disjuncts and each item in a disjunct is either an atomic formula, a negated atomic formula or a universally quantified sentence. Further processing is done separately of each disjunct and if all disjuncts are refuted, the inference is accepted.

Equalizer

EQUALIZER processes ground terms occurring in a disjunct. All ground terms are collected and then 'aggregate constants' are built by processing equalities among ground terms. An aggregate constant is an equality class formed in such a way that reflexivity, symmetry and transitivity of equality is taken into account. An aggregate constant remembers all its possible forms including their types. The following information is taken into account when building the aggregates:

- explicit equalities occurring in the disjunct;
- equalities inherited from the context of the inference as a result of processing the `take` and `reconsider` constructs;
- if an explicit equality of two structures occurs in the disjunct then the equalities of their selectors are added;
- properties of functors occurring in ground terms like commutativity or idempotency;
- conditional clusters are 'rounded-up' and as a result new adjectives may be added to an aggregate;
- requirements specified in the local environment. Requirements provide access to built-in features which can be selectively imported and some provide equalities, for example with requirements `REAL` the EQUALIZER uses the fact that that `x-0=x`.
- the values of integer expressions involving only small integer constants (up to 16 bit) and basic arithmetic operations are computed.

Aggregate constants are later used for substitutions in the UNIFIER.

EQUALIZER also adds new inequalities on the basis of the so called one difference rule. Given an inequality of two terms, a new inequality between their sub-terms is added provided it is the only inequality explaining the inequality of the original terms. Similarly, if an atomic sentence and a negation of an atomic sentence occurring in the disjunct differ only at one of their arguments, then a new inequality is added.

After creating aggregates and adding new inequalities, EQUALIZER tests whether a contradiction has been reached, that is, whether there is an inequality of two terms in the same aggregate.

Unifier

At this point, a disjunct to be refuted consists of a list of atomic formulae (but no equalities), negated atomic formulae and universal quantifiers. (If there are no universal sentences, the inference is not accepted.) All ground terms have been put into aggregate constants.

UNIFIER processes universal sentences occurring in a disjunct by considering one of them at a time. The universal sentence being processed is called the main premise; the remaining members of the disjunct are called auxiliary premises. The external universal quantifiers in the main premise are removed and their bound variables are replaced by free variables. The name UNIFIER is unfortunate as no unification is performed, only substitutions of aggregate constants for free variables are considered later.

What remains of the main premise now is best looked at as a `&` and `or` tree (`or` is internally represented as a negated conjunction). Because of de Morgan laws we can imagine that negations are pushed down to the leaves and each leaf is either an atomic formula, a negated formula, a universal formula or a negated universal (existential) formula. The leaf formulae are treated as a whole, no substitutions for bound variables of quantified leaf formulae are considered. For every leaf of the tree, UNIFIER looks for a complementary (with different sign) auxiliary premise with the same structure, that is having a ground term in a place where the leaf has a free variable. This way, a list of possible substitutions is built for every leaf (partial maps from free variables to ground terms) such that if they are applied, a contradiction results between a leaf and an auxiliary premise.

An algebra of substitutions is used to compute the substitutions for which the entire main premise is contradictory with the auxiliary premises. If the set of such substitutions is non empty the disjunct has been refuted.

4 Developing an article

A typical 'loop' when writing a MIZAR article is illustrated in Figure 2. An author prepares their text and presents it to the MIZAR processor (in our slang: one mizars a text). The processor creates a number of auxiliary files and also generates an error file. An additional utility program (error-printer) inserts the error messages into the source text. Now, the author tries to fix the errors and mizars the text again. If there are no changes in the environment part then the ACCOMMODATOR is not run. This loop is repeated until there are no errors.

Once we have created a satisfactory environment, further work with the MIZAR verifier is best described as a low intensity interaction: the entire text is processed each time. Such a low intensity interaction has its advantages as it stresses the division of responsibilities between the author and the checker ([10, 12]). The part of the MIZAR language for structuring proofs is close to mathematical vernacular and can be used to record just an idea of a proof (with many unproven steps). Then one can check the overall proof structure while postponing the checking of low-level details. Since the author works with the text, it is the author that is in control of this step-wise refinement process and thus can attempt a forward proof, or a backward proof, or any mixture of the two. Typically, a forward proof is attempted, if this fails, we try the proof backwards and then iterate.

MIZAR proofs are written in declarative style (see [12]), i.e. an author is writing a proof rather than some instructions to a machine of how to construct or reconstruct a proof which

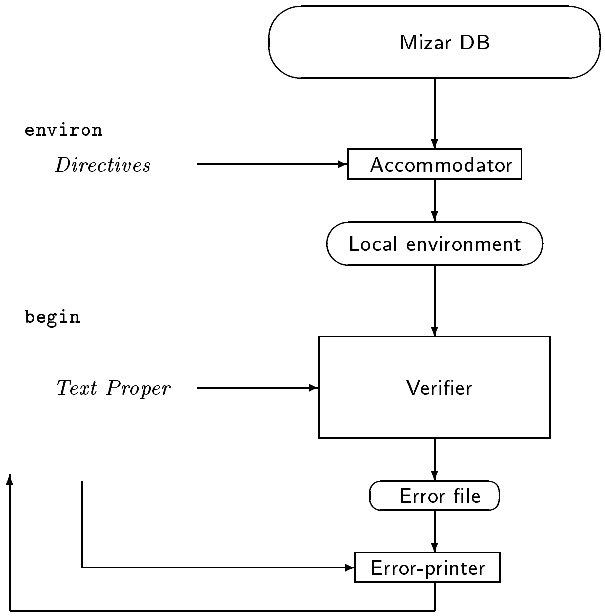

Figure 2: *Developing an article* loop

is the essence of the so called procedural style. This is important for the Mizar author: at all times, the author has access to and manipulates "a state" of the proof being developed and this "state" is recorded in the Mizar language. This is not so in the procedural style of proving when "a state" of a partial proof can be displayed albeit usually not in the same notation as the proof instructions given by the proof developer.

There are means for excluding proofs from being checked by replacing `proof` with `@proof`. This is employed for two reasons:

- In step-wise refinement we leave some proofs unfinished for future development.
- We exclude checking proofs which have been already completed.

The final check always consists of checking the entire article (with no `@proof`).

A natural question arises: how many times this development loop is repeated? This depends on the nature of the article and on the experience of the author. The less experienced authors tend to start writing a Mizar text while not having a clear idea of how to do a proof on paper. This compounded with their lax knowledge of Mizar results in running the Mizar processor many, many times and finally asking a more experienced Mizar user for help.

4.1 Errors

The errors reported by the Mizar processor can be roughly categorized as follows:

- **Lexical errors**: announced for undefined tokens.
- **Syntax errors**: the text is not written according to the Mizar grammar or an identifier is not defined; these errors are easy to correct.

- **Library directives errors**: importing from the library imposes certain conditions, for instance, when we want to import notations from an article then our local environment must include constructors to understand these notations. If this is not the case we get *Nothing imported from notations* error. Adding an appropriate constructors directive remedies the problem.

- **Constructor identification errors**: these errors are caused by the inadequacy of the local environment; for instance, we want to use a functor and the local environment does not include its definition. In the presence of overloading these errors are frequently difficult to remedy; and the remedy is an adjustment of the local environment.

- **Proof or diffuse statement structure**: The structure of the proof is guided by the sentence being proven and these errors indicate disagreements. A diffuse statement is like a proof-body without announcing a priori what is being proven. Its structure is similar to proof structure and similar errors occur.

  The first step in writing a proof is typically writing down its skeleton and having it accepted by the VERIFIER. This is a routine work to the point that it could be mechanized. This resembles step-wise refinement, as after completing the skeleton we have a number of 'smaller' claims to prove; smaller in that we have same additional assumptions to use. A proof by contradiction is a proof when at certain point we assume the negation of what remains to be proven and then we have to conclude a contradiction.

- **Checker errors**: Basically there is only one such error *This inference is not accepted* which is announced for *Straightforward Justifications*, see Section 3. This error does not mean that an inference step is logically invalid, it means that the MIZAR checker does not accept the step. It may be too complicated for the checker to see its correctness, and frequently such an inference step needs to be split into a sequence of 'smaller' steps. However, the most frequent cause of this error is an omitted premise.

- **Schemetizer errors**: a scheme is erroneously used.

- **Implementation restriction**: there is a large number of such errors due to assumed maximum sizes of machine representations of MIZAR constructs; for instance *Too long universal prefix*.

4.2 Constructor identification errors

Even for advanced MIZAR users, the most troublesome activity is preparing the environment directives, i.e. setting up the imports from the MML. Doing it well requires an intimate knowledge of MML and the current organization of MML seems not very well suited for this task. The difficulty stems from the fact that relevant material may be spread over a host of MIZAR articles and the only searching tools available at the moment are utilities of the *grep* family. Knowing the data base is the essential asset. The grep-like search is done over the MIZAR abstracts (see Section 5) and they constitute more than 7MB at the moment. On the current machines a run of grep over all abstracts takes only few seconds.

Consider the following example:

```
environ
 vocabulary FUNC;
 notation ARYTM, FUNCT_2, FUNCT_1;
 constructors ARYTM, FUNCT_2;
 clusters ARYTM;
 requirements SUBSET, ARYTM;
begin
 consider f being Function of NAT, NAT;
 consider g being Function;
 consider x being set;
 f.1 in NAT;
 g.x is set;
```

All the above directives are needed for the following reasons:

- vocabulary `FUNC` for mode name `Function` and operation symbol `.` used for function application.

- notation `ARYTM` for definition of the set of natural numbers `NAT`; notation `FUNCT_2` for definition of mode `Function of` (a function from a set to a set) and the function application functor `.` for functions of this type; notation `FUNCT_1` for definition of mode `Function` and the function application functor `.` for functions of this type; note that the functor symbol `.` is overloaded.

- constructors from `ARYTM` for requirements `ARYTM`, see below; constructors from `FUNCT_2` for the notation imported from `FUNCT_2`; note that we do not mention constructors from `FUNCT_1`—they are imported automatically as they are needed by constructors imported from `FUNCT_2`.

- clusters from `ARYTM` to know that `NAT` is non empty;

- requirements `SUBSET` and `ARYTM` to know that `1` belongs to `NAT`.

Note that due to the definition of the function application functor in `FUNCT_1` we can say that `g.x is set` even when we do not know whether `x` is in the domain of g.

If for some reason, the directives also include clusters from `RELSET_1`, then we get a checker error:

```
environ
 vocabulary FUNC;
 notation ARYTM, FUNCT_2, FUNCT_1;
 constructors ARYTM, FUNCT_2;
 clusters ARYTM, RELSET_1;
 requirements SUBSET, ARYTM;
begin
 consider f being Function of NAT, NAT;
 consider g being Function;
 consider x being set;
 f.1 in NAT;
 ::>      *4
 g.x is set;
 ::>
 ::> 4: This inference is not accepted
```

This should not be surprising! We have now a different environment. Due to the clusters imported from `RELSET_1` the mode `Function of` now widens to the mode `Function`; without these clusters it was not so and in `f.1` the function application was as defined for the mode `Function of` and in `g.x` as defined for the mode `Function`. Now, `f` has also the mode `Function`, so the question is which function application is to be used for `f`? This is resolved by trying available notations from `notation` in order listed and using the last one that fits. In this case, it is the function application as defined in `FUNCT_1` and this is not what we want. If we swap the order of imported notations, we get the expected result:

```
environ
 vocabulary FUNC;
 notation ARYTM, FUNCT_1, FUNCT_2;
 constructors ARYTM, FUNCT_2;
 clusters ARYTM, RELSET_1;
 requirements SUBSET, ARYTM;
begin
 consider f being Function of NAT, NAT;
 consider g being Function;
 consider x being set;
 f.1 in NAT;
 g.x is set;
```

The work on better organizing MML and improving the means for building the local environment is being continued.

4.3 Proof structure and checker errors

In the following example, we deal with an empty environment. This means that all what we can do is some 'gymnastics' in logic.

```
environ
begin
 scheme T{P[set],Q[set],R[set]} :
  for a being set st P[a] holds R[a]
 provided
 A: for a being set st P[a] holds Q[a] and
 B: for a being set st Q[a] holds R[a]
 proof          :: thesis is : for a being set st P[a] holds R[a]
  thus thesis by A, B;
::>            *4
end;
::>
::> 4: This inference is not accepted
```

The checker announces an error as it cannot see that the formulae labeled `A` and `B` imply the claim. We have to structure the proof in order to succeed. First we make a mistake in the proof structure:

```
environ
begin
 scheme T{P[set],Q[set],R[set]} :
  for a being set st P[a] holds R[a]
 provided
```

```
 A: for a being set st P[a] holds Q[a] and
 B: for a being set st Q[a] holds R[a]
 proof           :: thesis: for a being set st P[a] holds R[a]
  let a be set;  :: thesis: P[a] implies R[a]
  assume Q[a];
::>    *52
  hence thesis by B;
 end;
::>
::> 52: Invalid assumption
```

Certainly we are not allowed to assume `Q[a]` when proving the implication `P[a] implies R[a]`.

```
environ
begin
 scheme T{P[set],Q[set],R[set]} :
  for a being set st P[a] holds R[a]
 provided
 A: for a being set st P[a] holds Q[a] and
 B: for a being set st Q[a] holds R[a]
 proof           :: thesis: for a being set st P[a] holds R[a]
  let a be set;  :: thesis: P[a] implies R[a]
  assume P[a];   :: thesis: R[a]
  hence thesis by A, B;
::>             *4
 end;
::>
::> 4: This inference is not accepted
```

The checker is still unhappy, we have to make two steps in our reasoning.

```
environ
begin
 scheme T{P[set],Q[set],R[set]} :
  for a being set st P[a] holds R[a]
 provided
 A: for a being set st P[a] holds Q[a] and
 B: for a being set st Q[a] holds R[a]
 proof           :: thesis: for a being set st P[a] holds R[a]
  let a be set;  :: thesis: P[a] implies R[a]
  assume P[a];   :: thesis: R[a]
  then Q[a] by A;
  hence thesis by B;
 end;
```

One can get an impression that MIZAR checker is rather weak and thus forces annoyingly detailed reasonings, see Section 3. However, we have observed that the new MIZAR users tend to write their proofs at even more detailed level than required by the checker. Finding a right balance between what should be the responsibility of the user and what should be obvious to the checker is important and only extensive practice can indicate the direction towards a satisfactory solution.

Strengthening the checker such that the last proof would not require two inner steps is certainly possible. However, we are facing here a delicate trade-off. A more powerful checker will certainly run longer on more complicated inference steps and thus the time of checking an article would increase. Since the checker is run many, many times this would imply an idle time for the author. Since every author is interested in optimizing their time, it seems more reasonable that for batch processing we have a fast checker rather than a powerful and a slow one. Certainly, if a fully interactive proof assistant for MIZAR is built one can consider employing different decision procedures.

5 Including an article into MML

Once an article is completed and the VERIFIER finds no errors in it, the article may be presented for inclusion into the central Mizar Mathematical Library (MML). The decision about acceptance/rejection of an article is made by the Library Committee. At the moment there is no refereeing process besides mechanically checking the correctness of the article and running some mechanical 'improvers'. The need for human refereeing of articles may become a pressing issue with the growth of MML.

It seems crucial, at least at this stage of the system development, to maintain a centralized data base of articles. The MIZAR language and the checking software evolve and as a result the articles stored in MML need to be revised in order to take advantage of the new language features and the new capabilities of software. It is also the role of the Library Committee to perform such revisions.

An article accepted into MML gets a unique identifier and is processed into several forms later distributed together with the checking software:

- The source text of the article submitted by the author(s) is subjected to a number of utility programs which aim at discovering parts of the text that can be irrelevant and then some minimal automatic 'beautification' at the textual level. This source text may change in the future when the library undergoes a revision.

- An *abstract* of the article is a text file containing only the statements of theorems, definitions and schemes and they are uniformly labelled. These labels are used for referencing items in the article from other articles.

- *Data base* files (also called *library* files) which are used by ACCOMMODATOR when importing items from the article to the local environment of another article. These files are not meant to be read by MIZAR authors.

The *Text Proper* of an article contains *public* and *non-public* information. The public information are statements of: theorems, definitions and schemes. Their justifications and all other items from *Text Proper* are non-public and they leave no trace in either the abstract or the library files.

The process of creating the data base files for a new article included into MML is illustrated in Figure 3. The ACCOMMODATOR program is central in this process and the design of its functionality raises several interesting problems which we discuss in Section 6. This is the same program which is used when creating the local environment of an article. It creates a number of working files (not meant to be read by MIZAR authors) which are used when mizaring an article. From these files the EXTRACTOR program creates the data base files about the exported items: constructors, notation, clusters, theorems, definitions and schemes.

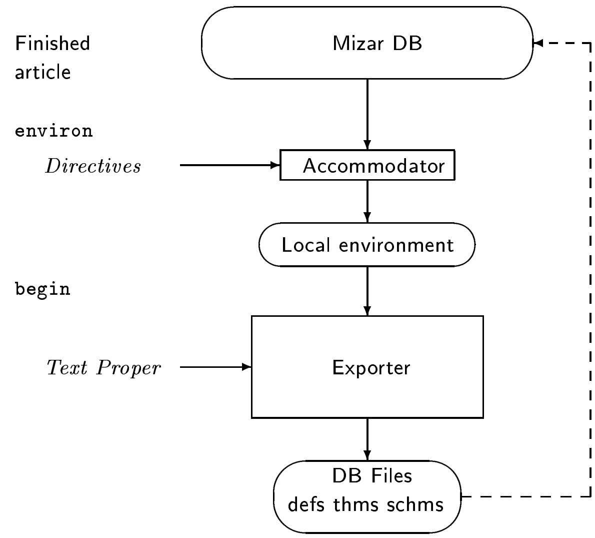

Figure 3: Including an article to MML

The library files of the public data base are built from the articles submitted to the MML and are distributed by the Mizar Society. A private data base may be created by a user (or a group of users) using the Exporter on articles stored in a private library. Private data bases have temporary character: once the author is sure of the quality of an article (which usually requires writing a couple of other articles using it), the article is submitted to the MML.

5.1 *Journal of Formalized Mathematics*

Every article accepted to MML gets its presence on the web in the electronic *Journal of Formalized Mathematics* maintained at `http://mizar.org/JFM`. The *Journal* is organized into annual volumes and each article is presented in a number of forms:

- A short summary provided by the author(s) in English.
- The abstract of the article in the html format where all defined items used in the article are hyperlinked to their place of definition. This browsing facility is very useful especially to the new authors.
- The full text of the article.
- The abstract of the article is automatically converted into TEX and the postscript version is posted.

As MML is continually revised, the electronic *Journal* is updated and presents the up to date version of the articles. There is an associated paper publication called *Formalized Mathematics* which publishes the articles in the state of their inclusion to MML. *Formalized Mathematics* is now published by University in Białystok, Poland; the initial volumes were published by Fondation Philippe le Hodey, Brussels, Belgium.

6 Local environment

The local environment of an article forms the connection between the article being developed and the MML data base. Why do we need it? Typically, when writing a mathematical paper we are focused on a narrow domain and we do not reference a big variety of other areas.

In a naive approach one might suggest that there is no local environment and the context in which an article is being checked is just the entire data base. While this may be a solution when the data base is very small, with growth of the data base we unavoidably run into the problem of size. Additionally, in the presence of symbol overloading we may face difficulties even with grammatical analysis of the source text. It seems awkward to require that every notion introduced is expressed by a unique notation.

Another extremal approach to creating the local environment is importing all needed items individually. This approach would require that every data base item is (uniquely) named. While this approach seems possible, it may turn out to be too tedious in practice (but only a large scale experiment could tell). One can modify this approach such that when we import a notion, say a functor, we recursively import all notions needed to represent also the types of its arguments and the type of its result. But here we face a problem: when shall we stop this recursive descent? Are we unfolding down to the basic notions? If so, then with a sizeable data base we may run again into the problem of size. Alternatively, we may look for some rules guiding and restricting this process of recursive imports.

We feel that the description of the local environment should be 'rough' meaning that we should not be forced to give too many details. This description certainly depends on the organization of the data base and with the current MML being a collection of articles, an article is a unit of import. The creation of the local environment is ruled by the imports of constructors according to the `constructors` directives. When we request constructors from an article, all constructors from this article are imported into the local environment. But that is not all. The ACCOMMODATOR after importing a constructor brings into the local environment also all the constructors needed to understand the arguments of the constructors already in the local environment. This process is continued recursively at the level of articles, that is, if a constructor from an article needs to be imported, the ACCOMMODATOR imports all the constructors from the article. Here we face the issue of how long this process should be continued. With a larger data base we may face the problem of too big an environment that is created and many of the imported constructors may be not needed for processing of the article being developed. We call this problem the "cutting-off" level of constructor imports. At which level we should stop this recursive process is a subject of current research.

The processing of remaining environment directives for importing notation, theorems, clusters, and schemes is driven by the collection of constructors already in the local environment. These imports are done by individual items; that is, if we request notation from an article, only these notation items are imported for which all the needed constructors are in the local environment. This aims at keeping the local environment small.

Certainly, a collection of source articles is not the only possible organization of a mathematical data base. One can imagine treating the source articles as "raw" material and creating topical articles on top of them. Then the the local environment can be described in terms of this topical higher level articles. However, the problem of keeping the local environment small still remains. For example, consider a topical unit which covers finite sequences (even now, all the facts about and tools for dealing with finite sequences form a sizeable chunk of MML spread over many articles). It is hard to imagine, that someone would need all this information in a local environment when proving a theorem. So the problem, of the "cutting-off" level of imports still remains as the size of the local environment has a crucial effect on resources, both time and speed, when processing an article under development.

It is felt, that with the small size of the MML data base now, the problems mentioned above may not be so burning. However, we should bear in mind that thinking of a date base for 'all' of mathematics we will have to deal with a data base quite a few orders of magnitude bigger.

7 Miscellany

Classical vs abstract mathematics

The current MML can be roughly divided into two parts:

- mathematics not using the concept of *structure*, we propose to call this part *classical mathematics*, and

- mathematics using the concept of *structure* which can be called *abstract mathematics*.

Undoubtedly, there are results belonging to classical mathematics which are best achieved using structures. For example, it would be hard to imagine a *reasonable* elimination of the notion of structure from the proof of Fermat Last Theorem. (On the other hand, structures can always be eliminated.) This type of results probably deserves a name *advanced mathematics*.

A typical example of classical mathematics in MML is arithmetic. The real numbers are defined without introducing the field of reals. The real numbers in MML are first introduced as a meta-structure, the structure is embedded into MML. We have the set of real numbers, their addition, multiplication and ordering. As long as we work with a specific structure, a meta-structure suffices. Once we want to develop a general theory and talk about relationship among structures, then the notion of a structure is indispensable. (In MML, the field of reals is defined later in [15].) Another examples of widely use meta-structures in MML: integer arithmetic, natural arithmetic, finite sets, finite sequences.

Choosing notions

The roles of classical and abstract mathematics are different. The former plays the role of a tool-kit, the latter addresses abstract problems. To use an analogy with computer software: classical mathematics is what corresponds to system software or firmware, abstract mathematics—to problem or user oriented software. The firmware should be compact, efficient and frugal, even at the expense of using some tricks.

Let us look at some examples illustrating the problems of choosing appropriate notions for the mathematical tool-kit. At the moment there are two notions of a pair in MML:

1. `[x,y]` defined as `{{x,y},{x}}`, that is the Kuratowski pair (see Section A) and

2. `<*x,y*>` defined as an `Element of 2-tuples_on` a set, that is a finite sequence of length 2 ([5]).

and there is a prevailing opinion that only one of them should be widely used: preferably the second, while the first should be avoided. Note that the first definition is used when defining relation, which is used to define function, which is needed to define finite sequence, so the first definition cannot be eliminated.

If we wanted to use an abstract notion of a pair then we would have to define a structure like:

```
definition
  let S be 1-sorted;
  struct (1-sorted) PairStr over S
    (# carrier -> set,
       Pair -> Function of [:the carrier of S, the carrier of S:], the carrier,
       Left, Right -> Function of the carrier, the carrier of S
    #)
```

and then for a pair we would use `Element of P` where `P` is a `PairStr`. Note however, that such a definition uses the notions of a Cartesian product and a function, but in order to define them we need some notion of pairing.

As another example let us take the notion of Cartesian product. A binary Cartesian product can be expressed either as:

1. `[:X,Y:]` defined in [6], or

2. `product <*X,Y*>` using the generalized product defined in [1].

(If we want a Cartesian square, we can use the above as `[:X,X:]` or `product <*X,X*>`; the latter has a semantically equal counterpart available through a different notation: `2-tuples_on X`, see [5].)

The question is whether we can justify maintaining these possibilities which were added to MML by various authors. It is typical in MML that when defining Cartesian products of structures, authors separately define a binary Cartesian product of structures and only then a general product (see products of universal algebras in [16] or products of many sorted algebras in [17]). This is repetition of effort and if we decide to use the second variant for the binary Cartesian product then a lot of duplicated material could be eliminated. This type of issues are subject to frequent MML revisions in order to maintain the integrity of the mathematical tool-kit.

How general?

There are many mathematical structures that are sufficiently important to develop their theories individually—groups, lattices etc. In order to develop a general theory of structures one must introduce a general notion of structure (for example a la Bourbaki) and then it would be possible to introduce a general notion of homomorphism, of substructure etc. in MML, this has been done for one-sorted algebras [14]. Since this is a simple case let us examine it a bit closer. The notion of a group may be defined in a number of ways using various constructs of Mizar or we can treat a group as a universal algebra. In the former case, we have to introduce all the needed notions from scratch (see [24]), in the latter, one can employ the notions (e.g. isomorphism) previously developed for universal algebras. Certainly, using a more general apparatus (universal algebras) may cause various troubles. However, by properly developing a group theory as a universal algebra one may be as comfortable working with such groups as with groups defined in a traditional way.

Why not work with groups defined from universal algebras? First of all, let us not forget other possible generalizations: e.g. introduce the notion of a group in a category. Developing the group theory in this approach yields the common group theory for the category of sets and for the category of topological spaces we get the theory of topological groups. One may foresee big gains as we would have a common theory for various groups: algebraic, metric, ordered, rings - as a group in the category of monoids, and so on. Certainly it is also possible to introduce the notion of universal algebra in a category and treat a group as a special case of such an algebra. Categories can be generalized and we may, for instance, consider such categories in which hom-sets are not just sets but objects of some closed category. Generalizations may go ad infinitum, and unlike many other cases where we have *one, two, three, heck of a lot* we are dealing here with a genuine infinity. Nonetheless, in order to do something specific, one has to stop at a certain point.

In MML we have the notion of a real vector space (see the series `RLVECT`) and more generally of a vector space over an arbitrary field (see the series `VECTSP`). Should we define complex vector spaces separately like the real vector space or employ the general notions from `VECTSP`?

Language evolution

At the moment, the Mizar language does not provide means to define the general notion of isomorphism for Mizar structures. In order to define such a notion, we would need variables ranging over structures, therefore we would need some type `Structure` and so on. In other words: the Mizar language would have to be generalized, but it boils down to developing some theory of structures (set theoretical) and possibly building it into the Mizar processor. This can be done, however, doing so will tempt implementing other generalizations that will inevitably show up and as a result we will have a rapid evolution of the language instead of developing the data base.

The above concerns what we call the *roofing* problem: constructs of the Mizar language are kept fixed in one place—let us call it a 'roof', and if anyone wants to look at their objects from a higher level, they must dig a few levels underground, put their objects there and then investigate them in the comfortable conditions of the ground level — the Mizar language.

Of course, the Mizar language has evolved in the past. This evolution has been slow and has been driven by the development of MML.

8 Conclusions

We would like to offer only one conclusion: we need more experience in building a data base of computer verified mathematics. Building such a data base is a substantial undertaking of evolving nature. We need to involve mathematicians—this seems to be a necessary condition for a success of such a project.

References

[1] Grzegorz Bancerek. König's theorem. *Formalized Mathematics*, 1(**3**):589–593, 1990.

[2] Grzegorz Bancerek. Zermelo theorem and axiom of choice. *Formalized Mathematics*, 1(**2**):265–267, 1990. `http://mizar.org/JFM/Vol1/wellord2.html`.

[3] Grzegorz Bancerek. On powers of cardinals. *Formalized Mathematics*, 3(**1**):89–93, 1992. `http://mizar.org/JFM/Vol4/card_5.html`.

[4] T. Becker and V. Weispfenning. *Groebner Bases.* Springer, 1993.

[5] Czesław Byliński. Finite sequences and tuples of elements of a non-empty sets. *Formalized Mathematics*, 1(**3**):529–536, 1990.

[6] Czesław Byliński. Some basic properties of sets. *Formalized Mathematics*, 1(**1**):47–53, 1990. `http://mizar.org/JFM/Vol1/zfmisc_1.html`.

[7] M. Davis. Obvious logical inferences. In *Proceedings of the 7th IJCAI*, pages 530–531, 1981.

[8] A. Degtyarev, A. Lyaletski, and M. Morokhovets. Evidence algorithm and sequent logical inference search. In *LNAI* ***1705***, pages 44–61, 1999.

[9] F. B. Fitch. *Symbolic Logic, An Introduction.* The Ronald Press Company, 1952.

[10] H. Friedman. FOM: 107: Automated proof checking, May 2001. `www.math.psu.edu/simpson/fom/postings`.

[11] G. Gierz, K. H. Hofmann, K. Keimel, M. Lawson, J. D. a nd Mislove, and D. S. Scott. *A Compendium of Continuous Lattices.* Springer-Verlag, 1980.

[12] John Harrison. Proof style. In Eduardo Giménex and Christine Pausin-Mohring, editors, *Types for Proofs and Programs: International Workshop TYPES'96*, volume 1512 of *LNCS*, pages 154–172, Aussois, France, 1996. Springer-Verlag.

[13] S. Jaśkowski. On the rules of supposition in formal logic. *Studia Logica*, 1, 1934.

[14] Małgorzata Korolkiewicz. Homomorphisms of algebras. Quotient universal algebra. *Formalized Mathematics*, 4(**1**):109–113, 1993.

[15] Eugeniusz Kusak, Wojciech Leończuk, and Mic hał Muzalewski. Abelian groups, fields and vector spaces. *Formalized Mathematics*, 1(**2**):335–342, 1990.

[16] Beata Madras. Product of family of universal algebras. *Formalized Mathematics*, 4(**1**):103–108, 1993.

[17] Beata Madras. Products of many sorted algebras. *Formalized Mathematics*, 5(**1**):55–60, 1996.

[18] S. McCall, editor. *Polish Logic in 1920–1939.* Clarendon Press, Oxford, 1967.

[19] M. Muzalewski, editor. *An outline of PC-Mizar*. Fondation Philippe le Hodey, Brussels, 1993. www.cs.ualberta.ca/~piotr/Mizar/Doc/MM-manual.ps.

[20] K. Ono. On a practical way of describing formal deductions. *Nagoya Mathematical Journal*, 21, 1962.

[21] H. Rasiowa and R. Sikorski. *The Mathematics of Metamathematics*. PWN, Warszawa, 1968.

[22] P. Rudnicki and A. Trybulec. On Equivalents of Well-foundedness. An experiment in MIZAR. *Journal of Automated Reasoning*, 23:197–234, 1999.

[23] Andrzej Trybulec. Tarski Grothendieck set theory. *Formalized Mathematics*, 1(**1**):9–11, 1990. http://mizar.org/JFM/Axiomatics/tarski.html.

[24] Wojciech A. Trybulec and Michał J. Trybulec. Homomorphisms and isomorphisms of groups. Quotient group. *Formalized Mathematics*, 2(**4**):573–578, 1991.

[25] F. Wiedijk. Mizar: an impression. www.cs.kun.nl/~freek/mizar/mizarintro.ps.gz.

A MML Axiomatics

The development of mathematics in Mizar is based on a version of the Zermelo-Fraenkel set theory.

Several notions of set theory are built into the Mizar processor and their 'user interface' is given in a special article named `HIDDEN`. The symbols used in defining these notions are given in a special vocabulary also called `HIDDEN`. The conceptual framework introduced in `HIDDEN` is automatically added to the local environment of each article checked by the Mizar processor (and so are the symbols from vocabulary `HIDDEN`).

The first notion defined in `HIDDEN` is a mode `set`:

```
 definition
 mode set;
end;
```

This mode is the root of the Mizar type hierarchy; the type of every object ultimately widens to `set`.

Absolute equality is built-in as a reflexive, symmetric and transitive predicate, written in the common infix notation.

```
 definition let x,y be set;
 pred x = y;
   reflexivity;
   symmetry;
  antonym x <> y;
end;
```

The above definition also introduces the inequality predicate as an antonym of equality, so that `x <> y` can be written instead of `not x = y`. Note that transitivity is not announced in this definition but equality is processed as a transitive predicate.

The asymmetric elementhood predicate `in` is built-in:

```
 definition let x,X be set;
 pred x in X;
  asymmetry;
end;
```

From the asymmetry we have that `not x in x`.

The inclusion predicate is given its syntax here:

```
 definition let X,Y be set;
 pred X c= Y;
   reflexivity;
end;
```

and its reflexivity is announced; the definiens is given in a redefinition of the predicate in `TARSKI` [23] (see below).

The axioms of set theory are contained in article `TARSKI` [23] which is the only MIZAR article not checked for correctness (`HIDDEN` plays an auxiliary role).

- **Extensionality axiom**

```
theorem                                                                   :: TARSKI:2
(for x holds x in X iff x in Y) implies X = Y;
```

- **Singleton and doubleton axioms**

```
 definition let y; func { y } -> set means                              :: TARSKI:def 1
 x in it iff x = y;
 let z; func { y, z } -> set means                                      :: TARSKI:def 2
  x in it iff x = y or x = z;
 commutativity;
end;
```

  Note that the singleton could have been defined once the doubleton was available.

- Definition of set inclusion (not an axiom, a defined predicate):

```
 definition let X,Y;
 redefine pred X c= Y means                                             :: TARSKI:def 3
  x in X implies x in Y;
end;
```

- **Union axiom**

```
 definition let X;
  func union X -> set means                                             :: TARSKI:def 4
   x in it iff ex Y st x in Y & Y in X;
end;
```

- **Regularity axiom**

```
theorem                                                          :: TARSKI:7
x in X implies ex Y st Y in X & not ex x st x in X & x in Y;
```

- **Replacement axiom**

```
scheme Fraenkel { A()-> set, P[set, set] }:
ex X st for x holds x in X iff ex y st y in A() & P[y,x]
provided for x,y,z st P[x,y] & P[x,z] holds y = z;
```

- Ordered pairs (not an axiom, definable with singleton and doubleton)

```
definition let x,y;
func [x,y] equals                                            :: TARSKI:def 5
 { { x,y }, { x } };
end;
```

- Equipotence of sets (not an axiom, a defined predicate)

```
definition let X,Y;
pred X,Y are_equipotent means                                :: TARSKI:def 6
  ex Z st
     (for x st x in X ex y st y in Y & [x,y] in Z) &
     (for y st y in Y ex x st x in X & [x,y] in Z) &
      for x,y,z,u st [x,y] in Z & [z,u] in Z holds x = z iff y = u;
end;
```

- **Tarski axiom** on existence of arbitrarily large, strongly inaccessible cardinals.

```
theorem                                                          :: TARSKI:9
  ex M st N in M &
    (for X,Y holds X in M & Y c= X implies Y in M) &
    (for X st X in M ex Z st Z in M & for Y st Y c= X holds Y in Z) &
    (for X holds X c= M implies X,M are_equipotent or X in M);
```

Using this axiom one can define infinite sets (article `CARD_5` [3]), the powerset operator (article `ZFMISC_1` [6]), and prove the axiom of choice (article `WELLORD2` [2]). Note, that the axiom of choice can also be proven using the available counterpart of the ε operator.