Overview of the tutorial

- Part 1 (Adam Naumowicz): Introduction
- Part 2 (Artur Korniłowicz): Formalizing an example theory
- Part 3 (Adam Grabowski): Actual hands-on session
What is **Mizar**?

- **Mizar** is a system for formalizing and proof-checking mathematics invented by Andrzej Trybulec (†2013) and developed since 1970s.
- Its language tries to mimic standard mathematical practice.
- Its verification engine is designed to preserve human understanding of proof steps.
- It is being used to build a centralized library of formalized mathematical knowledge based on simple axioms (of set theory) - **Mizar** Mathematical Library (MML).
What do we mean by “formalizing mathematics” here?

- Precise meaning of every single notion used in mathematical text
  - Full disambiguation of used notions
  - Clear dependence of definitions, axioms and theorems
  - Rigorous use of deduction rules

- The formalization should be understandable for a computer system to automatically perform the following tasks:
  - Checking lexical and grammatical correctness
  - Linking new developments with the data already available
  - Verifying logical validity of all inference steps

- Ideally, the computer input language should facilitate various purposes of developing mathematical proofs
  - Convincing
  - Documenting
  - Presentation
Key features of the Mizar system

- The system uses classical first-order logic
- Statements with free second-order variables (e.g. the induction scheme) are supported
- The system uses natural deduction for doing conditional proofs
- The system uses a declarative style of writing proofs (mostly forward reasoning) - resembling mathematical practice
“A good system without a library is useless. A good library for a bad system is still very interesting... So the library is what counts.”
(F. Wiedijk, Estimating the Cost of a Standard Library for a Mathematical Proof Checker.)

- A systematic collection of articles started around 1989
- Recent MML version - 5.37.1275
  - includes 1275 articles written by over 250 authors
  - over 56000 theorems
  - over 11000 definitions
  - over 800 schemes
  - over 13000 registrations
- The library is based on the axioms of Tarski-Grothendieck set theory
The **Mizar** language

- The proof language is designed to be as close as possible to “mathematical vernacular” and be automatically verifiable
  - It is a reconstruction of the language of mathematics
  - It forms “a subset” of standard English used in mathematical texts
  - The language is highly structured - to ensure producing rigorous and semantically unambiguous texts
  - It allows prefix, postfix, infix notations for predicates as well as parenthetical notations for functors
The \textbf{Mizar} language - ctd. (1)

The language includes the standard set of first order logical connectives and quantifiers for forming formulas:

\begin{center}
\begin{tabular}{c|c}
$\neg \alpha$ & \textit{not $\alpha$} \\
$\alpha \land \beta$ & $\alpha \& \beta$ \\
$\alpha \lor \beta$ & $\alpha \text{ or } \beta$ \\
$\alpha \rightarrow \beta$ & $\alpha \text{ implies } \beta$ \\
$\alpha \leftrightarrow \beta$ & $\alpha \text{ iff } \beta$ \\
$\exists x \alpha$ & $\text{ex} \ x \text{ st } \alpha$ \\
$\forall x \alpha$ & $\text{for} \ x \text{ st } \alpha$ \\
$\forall x: \alpha \beta$ & $\text{for} \ x \text{ st } \alpha \text{ holds } \beta$
\end{tabular}
\end{center}
Each quantified variable has to be given its type, so the quantifiers actually take the form

\[
\text{for } x \text{ being set holds } \ldots
\]
or

\[
\text{ex } y \text{ being real number st } \ldots
\]

where set and real number represent examples of types.

Mizar allows to globally assign this type to selected variable names with a reservation

\[
\text{reserve } x, y \text{ for real number;}
\]

Then one does not have to mention the type of \( x \) or \( y \) in quantified formulas.

With some reservations declared, Mizar implicitly applies universal quantifiers to formulas if needed.
The formulas

\[ \text{for } x \text{ holds for } y \text{ holds ...} \]

or

\[ \text{for } x \text{ holds } \exists y \text{ st ...} \]

may be shortened to

\[ \text{for } x \text{ for } y \text{ holds ...} \]

and

\[ \text{for } x \text{ } \exists y \text{ st ...} \]

Instead of writing

\[ \text{for } x \text{ holds for } y \text{ holds ...} \]

or

\[ \exists x \text{ st } \exists y \text{ st ...} \]

more convenient forms with lists of variables are allowed

\[ \text{for } x, y \text{ holds ...} \]

and

\[ \exists x, y \text{ st ...} \]

The binding force of quantifiers is weaker than that of connectives.
The **Mizar** language - ctd. (4)

- **Mizar reserved words (please mind that the language is case-sensitive):**

  - according
  - attr
  - cases
  - constructors
  - defpred
  - from
  - if
  - means
  - or
  - @proof
  - requirements
  - st
  - the
  - uniqueness
  - aggregate
  - axiom
  - cluster
  - contradiction
  - end
  - func
  - iff
  - mode
  - otherwise
  - provided
  - reserve
  - struct
  - then
  - theorem
  - vocabularies
  - and
  - be
  - coherence
  - correctness
  - environ
  - given
  - implies
  - non
  - over
  - qua
  - sch
  - such
  - theorem
  - when
  - antonym
  - begin
  - commutativity
  - def
  - equals
  - hence
  - involutiveness
  - not
  - per
  - reconsider
  - scheme
  - suppose
  - theorems
  - where
  - as
  - being by
  - compatibility
  - deffunc
  - ex
  - hereby
  - irreflexivity
  - notation
  - pred
  - redefine
  - schemes
  - symmetry
  - thesis
  - with
  - associativity
  - canceled
  - connectedness
  - define
  - ex
  - exactly
  - holds
  - is
  - notations
  - prefix
  - reflexivity
  - section
  - synonym
  - thus
  - wrt
  - assume
  - case
  - consider
  - definition
  - existence
  - idempotence
  - it
  - list
  - projectivity
  - registration
  - selector
  - set
  - that
  - transitivity

---

Adam Naumowicz, Artur Korniłowicz, Adam Grabowski

Institute of Informatics, University of Białystok, Poland

*Mizar Hands-on Tutorial*
Mizar special symbols:

, ; : ( ) [ ] { } = & ->

.= ... $1$2 $3$4 $5$6 $7$8 $9$10 (# #)

- A double colon (::) in Mizar texts starts a one-line comment
- If the double colon is followed by the dollar sign ($), this makes a special pragma (e.g. ::$V$)
Is Mizar typed or untyped?

- In a foundational sense, Mizar is based on **untyped** set theory.
- No particular axiom system is imposed by the system (MML is based on Tarski-Grothendieck set theory).
- Its objects are “just one type” (no pre-imposed disjointness, inclusion, or similar conditions on these objects via a foundational mechanism decoupled from the underlying classical logic).

The objects can still have various properties (a number, ordinal number, complex number, Conway number, a relation, function, complex function, complex matrix) which require different treatment, so they must be **typed**.

- It is not enough to classify them into “sorts” or otherwise disjoint “kinds”, because we want them to represent various (dependent) predicates.
- Types are used in quantified and qualifying formulas, for parsing, semantic analysis, overloading resolution, and inferring object properties.
The type system can be characterized by:

- soft-typing with possibly “dynamic” type change,
- typing information in a syntactically “elegant” way (resembling mathematical practice, e.g. via using dependent types and attributes)
  - types can have an empty list of arguments (most commonly they have explicit and/or implicit arguments),
  - adjectives can also be expressed with their own visible arguments, e.g., n-dimensional, or X-valued
- types are non-empty by definition (to guarantee that the formalized theory always has some denotation).
Reconstructing the type system

There have been attempts to reconstruct elements of this type system in order to translate the mathematical data encoded in MML into

- common mathematical data exchange formats, e.g. OMDoc,
- other proof assistants, e.g. HOL Light or Isabelle.

A particular advantage of the soft-typing approach is its straightforward translation to first-order ATP formats (allows developing hammer-style ITP methods).
**Mizar glossary**

- **Formulae** are constructed with **predicates** and the constructors of **terms** are called **functors**.
- When any variable is introduced in Mizar, its **type** must be given (the most general type being **object**).
- For any term, the verifier computes its unique type.
- **Types** in **Mizar** are constructed using **modes** and the constructors of **adjectives** are called **attributes** (every attribute introduces two adjectives, e.g. **empty** and **non empty**).
- **Structures** (record types) and their fields are created with **structural modes** and **selectors**, respectively.
**Mizar** type constructors

**Mizar** supports two kinds of mode definitions:

1. modes defined as a collection (called a cluster) of adjectives associated with an already defined radix type to which they may be applied, called expandable modes,

   definition
   
   let G,H be AddGroup;
   
   mode Homomorphism of G,H is additive Function of G,H;
   
   end;

2. modes that define a type with an explicit definiens that must be fulfilled for an object to have that type.

   definition
   
   let G be AbGroup, K,L be Ring;
   
   let J be Function of K,L;
   
   let V be for LeftMod of K, W be LeftMod of L;
   
   mode Homomorphism of J,V,W -> Function of V,W means
   
   (for x,y being Vector of V holds it.(x+y) = it.x+it.y) &
   
   for a being Scalar of K, x being Vector of V holds it.(a*x) = J.a*it.x;
   
   end;
Examples of attributes

- Without implicit parameters:

  ```
  definition
  let R be Relation;
  attr R is well_founded means
  for Y being set st Y c= field R & Y <> {} ex a being set st a in Y & R-Seg a misses Y;
end;
  ```

- With an implicit parameter:

  ```
  definition
  let n be Nat, X be set;
  attr X is n-at_most_dimensional means
  for x being set st x in X holds card x c= n+1;
end;
  ```
Types of mathematical objects defined in the Mizar library form a sup-semilattice with widening (subtyping) relation as the order. There are two hierarchies of types:

1. the main one based on the type set, and
2. the other based on the notion of structure.

The most general type in Mizar (to which both sets and structures widen) is called object.
Mizar structural types

- Structures model mathematical notions like groups, topological spaces, categories, etc. which are usually represented as tuples.
- A structure definition contains, therefore, a list of selectors to denote its fields, characterized by their name and type.
- Mizar supports multiple inheritance of structures that makes a whole hierarchy of interrelated structures available in the library, with the 1-sorted structure being the common ancestor of almost all other structures.
- One can define structures parameterized by arbitrary sets, or other structures.

```mizar
definition
  let F be 1-sorted;
  struct(addLoopStr) ModuleStr over F
  (# carrier -> set,
    addF -> BinOp of the carrier,
    ZeroF -> Element of the carrier,
    lmult -> Function of [:the carrier of F, the carrier:], the carrier #);
end;
```

Adam Naumowicz, Artur Korniłowicz, Adam Grabowski
Institute of Informatics, University of Bialystok, Poland

Mizar Hands-on Tutorial
Type change mechanisms

The effective (semantic) type of a given Mizar term is determined by a number of factors - most importantly, by the available (imported from the library or introduced earlier in the same formalization) redefinitions and adjective registrations.

**Redefinitions** are used to change the definiens or type for some constructor if such a change is provable with possibly more specific arguments. Depending on the kind of the redefined constructor and the redefined part, each redefinition induces a corresponding correctness condition that guarantees that the new definition is compatible with the old one.

**Registrations** refer to several kinds of Mizar features connected with automatic processing of the type information based on adjectives. Grouping adjectives in so called clusters (hence the keyword cluster used in their syntax) enables automation of some type inference rules. Existential registrations are used to secure the nonemptiness of Mizar types. The dependencies of adjectives recorded as conditional registrations are used automatically by the Mizar verifier.
Example of a mode redefinition

- **Original definition:**
  
  ```
  definition
  let X;
  mode Element of X -> set means
  it in X if X is non empty otherwise it is empty;
  end;
  ```

- **A redefinition:**
  
  ```
  definition
  let A, B be non empty set;
  let r be non empty Relation of A, B;
  redefine mode Element of r -> Element of [:A,B:];
  end;
  ```
Example of an attribute redefinition

- **Original definition:**

```plaintext
definition
  let R be Relation;
  attr R is co-well_founded means
  R~ is well_founded;
end;
```

- **A redefinition:**

```plaintext
definition
  let R be Relation;
  redefine attr R is co-well_founded means
  for Y being set st Y c= field R & Y <> {} 
  ex a being object st a in Y & for b being object st b in Y & a <> b 
  holds not [a,b] in R;
end;
```
Examples of registrations

- **Existential:**
  ```plaintext
  registration
  let n be Nat;
  cluster n-at_most_dimensional subset-closed non empty for set;
end;
```

- **Conditional:**
  ```plaintext
  registration
  let n be Nat;
  cluster n-at_most_dimensional -> finite-membered for set;
end;
```

- **Functorial (term):**
  ```plaintext
  registration
  let n be Nat;
  let X, Y be n-at_most_dimensional set;
  cluster X \ Y -> n-at_most_dimensional;
end;
```
Explicit type change

- For syntactic (identification) purposes, e.g. to force the system use one of a number of matching redefinitions, the type of a term can be explicitly qualified to one which is less specific, e.g.
  
  $1$ qua real number

  whereas in standard environments the constant has the type natural number and then appropriate (more specific) definitions apply to it.

- The reconsider statement forces the system to treat any given term as if its type was the one stated (with extra justification provided), e.g.

  reconsider $R$ as Field

  whereas the actual type of the variable $R$ might be $\text{Ring}$. It is usually used if a particular type is required by some construct (e.g. definitional expansion) and the fact that a term has this type requires extra reasoning after the term is introduced in a proof.
During the proof-checking phase, Mizar uses a non-trivial dependent congruence-closure algorithm (Equalizer) that merges terms that are known to be semantically equal, merging also their (dependent) soft-types – occasionally deriving a contradiction from adjectives like “empty” and “non-empty” – and propagating such mergers up the term and type hierarchy.

The refutational Mizar proof checker takes advantage of this, by doing all its work on the resulting semantic aggregated equivalence classes of terms, each having many properties – “superclusters” derived by the type system and the congruence closure algorithm, i.e., by calculating a transitive closure of all available registrations over the merged terms.
Miscellaneous type system features

- The global choice construction, e.g. the natural number, allows to introduce the unique constants for each well-defined type.

- Selected types can have a special sethood property registered. This property means that all objects of the type for which the property is declared are elements of some set and in consequence it is valid to use them within a Fraenkel term (set comprehension) operator.

- The construction the set of all is an abbreviation for Fraenkel terms defining sets of terms where the terms do not have to satisfy any additional constraints, e.g. the set of all n where n is natural number.

- Selected types have extra processing in the Mizar verifier (switched on by the so called requirements directives) in order to automate some typical tasks and exploit their properties to make routine inferences obvious, e.g. the computational processing of objects whose type widens to the type complex number.
More language constructs (definitions)

- Synonyms/antonyms
- “properties”
  - E.g. commutativity, reflexivity, transitivity etc.
- “requirements”
  - E.g. the built-in arithmetic on complex numbers
- Identifying (formally different, but equal) constructors
- Reductions (to simpler forms built from their subarguments)
More language constructs (proofs)

- Fraenkel terms (set comprehension binders)
- Iterative equalities
- “Syntactic sugar” features
Approximating informal mathematics in **Mizar**

- **Formal proof sketches**
  - A *formal proof sketch* is a formalization which is
    - Shorter than the full formalization (details of justification are not presented)
    - It can be extended to full formalization (then it is *correct*)
    - There are matching locations in both versions (one could fold and unfold pieces of text between both versions)

- **In a general setting**
  - Encoding in the correct syntax
  - Leaving out references in inference steps

- **In the case of **Mizar**
  - Encoding with no Parser, Analyzer and Reasoner errors
  - Ignoring Verifier errors (*4 and *1)
For any formula $\Phi$ its proof may take the form of a proof block in which the same formula is finally stated as a conclusion after the $\texttt{thus}$ keyword.

\begin{verbatim}
\phi
proof
... 
thus $\Phi$;
end;
\end{verbatim}
If the formula to be proved is a conjunction, then the proof should contain two conclusions:

\[ \Phi_1 \& \Phi_2 \]

proof
...
  thus \( \Phi_1 \);
...
  thus \( \Phi_2 \);
end;
When proving an implication, the most natural proof is the one where we first assume the antecedent and conclude with the consequent:

\[ \Phi_1 \implies \Phi_2 \]

```plaintext
proof
  assume \( \Phi_1 \);  
  ...  
  thus \( \Phi_2 \);  
end;
```


Adam Naumowicz, Artur Korniłowicz, Adam Grabowski

Institute of Informatics, University of Bialystok, Poland

Mizar Hands-on Tutorial
Equivalence is interpreted as a conjunction of two implications, which yields the following proof skeleton:

\[ \Phi_1 \iff \Phi_2 \]

proof

... thus \( \Phi_1 \) implies \( \Phi_2 \);

... thus \( \Phi_2 \) implies \( \Phi_1 \);

end;
The level of proof nesting can be reduced using the following skeleton:

\[ \Phi_1 \text{ iff } \Phi_2 \]

proof
hereby
  assume \( \Phi_1 \);
...
  thus \( \Phi_2 \);
end;
assume \( \Phi_2 \);
...
  thus \( \Phi_1 \);
end;
Disjunction is usually proved by assuming that the first disjunct does not hold and then to proving the other:

\[
\Phi_1 \text{ or } \Phi_2
\]

proof

assume not \( \Phi_1 \);

... 

thus \( \Phi_2 \);

end;
Any formula can also be proved using the *reductio ad absurdum* method:

\[ \phi \]

```
proof
  assume not \phi;
  ...
  thus contradiction;
end;
```
A proof of a universally quantified formula starts with selecting an arbitrary but fixed variable of a certain type and then concluding the validity of that formula substituted with it:

```
for a being Θ holds Φ
proof
  let a be Θ;
  ...
  thus Φ;
end;
```
A proof of an existential statement must provide a witness term $a$ and an appropriate conclusion.

```plaintext
ex a being $\Theta$ st $\Phi$
proof

... take $a$;

... thus $\Phi$;
end;
```
The **Reasoner** module is responsible for checking if a proof tactic used by the author corresponds to the formula being proved. The checking is based on the internal representation of formulas in a simplified “canonical” form - their *semantic correlates* using only VERUM, not, & and for ... holds ... together with atomic formulas.

Other formulas are encoded using the following set of rules:

- VERUM is the neutral element of the conjunction.
- Double negation rule is used.
- De Morgan’s laws are used for disjunction and existential quantifiers.
- $\alpha$ implies $\beta$ is changed into $\neg(\alpha \land \neg \beta)$.
- $\alpha$ iff $\beta$ is changed into $\alpha$ implies $\beta$ & $\beta$ implies $\alpha$, i.e. $\neg(\alpha \land \neg \beta) \land \neg(\beta \land \neg \alpha)$.
- Conjunction is associative but not commutative.
Mizar checks all first order statements in an article for logical correctness using its Checker module equipped with a certain concept of obviousness of inferences (classical disprover). In that module an inference of the form

\[
\frac{\alpha_1, \ldots, \alpha_k}{\beta}
\]

is transformed to

\[
\frac{\alpha_1, \ldots, \alpha_k, \lnot \beta}{\bot}
\]
A disjunctive normal form (DNF) of the premises is then created and the system tries to refute it

\[ ([\neg \alpha^{1,1} \land \cdots \land \neg \alpha^{1,k_1}] \lor \cdots \lor ([\neg \alpha^{n,1} \land \cdots \land \neg \alpha^{n,k_n}] \downarrow \]

where \( \alpha^{i,j} \) are atomic or universal sentences (negated or not)
For the inference to be accepted, all disjuncts must be refuted. So in fact $n$ inferences are checked

\[ \neg \alpha^{1,1} \land \cdots \land \neg \alpha^{1,k_1} \]
\[ \vdash \]
\[ \vdots \]
\[ \neg \alpha^{n,1} \land \cdots \land \neg \alpha^{n,k_n} \]
\[ \vdash \]
A typical induction scheme in Mizar

- scheme :: NAT_1:sch 2
  NatInd { P[Nat] } : for k being Nat holds P[k]
  provided
  P[0] and
  for k being Nat st P[k] holds P[k + 1];
reserve $i,j,k,l,m,n$ for natural number;

$i+k = j+k$ implies $i=j$;
proof
  defpred P[natural number] means 
  $i+1 = j+1$ implies $i=j$;
A1: $P[0]$
proof
  assume B0: $i+0 = j+0$;
  B1: $i+0 = i$ by INDUCT:3;
  B2: $j+0 = j$ by INDUCT:3;
  hence thesis by B0,B1,B2;
end;
A2: for $k$ st $P[k]$ holds $P[\text{succ }k]$
proof
  let $l$ such that C1: $P[l]$;
  assume C2: $i+\text{succ }l = j+\text{succ }l$;
  then C3: $\text{succ}(i+l) = j+\text{succ }l$ by C2,INDUCT:4
  .= $\text{succ}(j+l)$ by INDUCT:4;
  hence thesis by C1,INDUCT:2;
end;
for $k$ holds $P[k]$ from INDUCT:sch 1(A1,A2);
hence thesis;
end;
Running the system

- Logical modules (passes) of the Mizar verifier
  - **Parser** (*Tokenizer* + identification of so-called “long terms”)
  - **Analyzer** (+ **Reasoner**)
  - **Checker** (**Preparator**, **Prechecker**, **Equalizer**, **Unifier**) + **Schematizer**

- Communication with the database
  - **Accommodator**
  - **Exporter** + **Transferer**
Running the system – ctd.

- The interface (CLI, Emacs **Mizar** Mode by Josef Urban, “remote processing”)
  - The way **Mizar** reports errors resembles a compiler’s errors and warnings
  - Top-down approach
  - Stepwise refinement
  - It’s possible to check correctness of incomplete texts
  - One can postpone a proof or its more complicated part
Enhancing Mizar texts

- Utilities detecting irrelevant parts of proofs
  - relprem
  - relinfer
  - reliters
  - trivdemo
  - ...

Importing notions from the library

- The structure of Mizar input files
  - environ
  - ....
  - begin
  - ....

- Library directives
  - vocabularies (using symbols)
  - constructors (using introduced objects)
  - notations (using notations of objects)
  - theorems (referencing theorems)
  - schemes (referencing schemes)
  - definitions (automated unfolding of definitions in Reasoner)
  - equalities (importing definitions of terms defined with equals into the Checker)
  - expansions (importing definitional theorems of predicates into the Checker)
  - registrations (automated processing of adjectives)
  - requirements (using built-in enhancements for certain constructors, e.g. complex numbers)
Miscelanea

- Formalized Mathematics - FM (http://mizar.org/fm)
- MMLQuery - search engine for MML (http://mmlquery.mizar.org)
- Mizar TWiki (http://wiki.mizar.org)
- Mizar mode for GNU Emacs (http://wiki.mizar.org/twiki/bin/view/Mizar/MizarMode)
- MizAR: parallelized AI/ATP, verification, and presentation service for Mizar (http://mizar.cs.ualberta.ca/ñptp/MizAR.html)
Recommended reading

- F. Wiedijk, Writing a Mizar article in nine easy steps. (http://www.cs.ru.nl/~freek/mizar/mizman.ps.gz)