A Guide to the Mizar Soft Type System

Adam Naumowicz\textsuperscript{1}  Josef Urban\textsuperscript{2}

Institute of Informatics
University of Białystok, Poland
adamn@mizar.org

Czech Technical University in Prague
Czech Republic
josef.urban@gmail.com

TYPES 2016, Novi Sad, May 23-26, 2016
What is **MIZAR**?

- **MIZAR** is a system for encoding 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).
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.
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).
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.

A. Naumowicz and J. Urban

*A Guide to the Mizar Soft Type System*
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).
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).
**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,

\[
\text{definition} \\
\quad \text{let } G, H \text{ be } \text{AddGroup}; \\
\quad \text{mode } \text{Homomorphism of } G, H \text{ is additive Function of } G, H; \\
\text{end;}
\]

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

\[
\text{definition} \\
\quad \text{let } G \text{ be } \text{AbGroup}, K, L \text{ be } \text{Ring}; \\
\quad \text{let } J \text{ be Function of } K, L; \\
\quad \text{let } V \text{ be for LeftMod of } K, W \text{ be LeftMod of } L; \\
\quad \text{mode } \text{Homomorphism of } J, V, W \rightarrow \text{Function of } V, W \text{ means} \\
\quad \text{for } x, y \text{ being Vector of } V \text{ holds } \text{it.} (x+y) = \text{it}.x + \text{it}.y \text{ &} \\
\quad \text{for } a \text{ being Scalar of } K, x \text{ being Vector of } V \text{ holds } \text{it.} (a*x) = J.a*\text{it}.x; \\
\text{end;}
\]
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;
The lattice of **MIZAR** types

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`.
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. \textit{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.

```plaintext
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;
```

A. Naumowicz and J. Urban
A Guide to the \textit{Mizar} Soft Type System
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:
  ```plaintext
  definition
    let X;
    mode Element of X -> set means
    it in X if X is non empty otherwise it is empty;
  end;
  ```

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

- **A redefinition:**
  ```
  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:

registration
let n be Nat;
cluster n-at_most_dimensional subset-closed non empty for set;
end;

■ Conditional:

registration
let n be Nat;
cluster n-at_most_dimensional -> finite-membered for set;
end;

■ Functorial (term):

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 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, \textsc{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 \textsc{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.
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.