A Guide to the Mizar Soft Type System

Adam Naumowicz[1] Josef Urban[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?

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.

MIZAR type system's main features

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

- 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 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;
```

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`.

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.

```
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;
```

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:

```
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

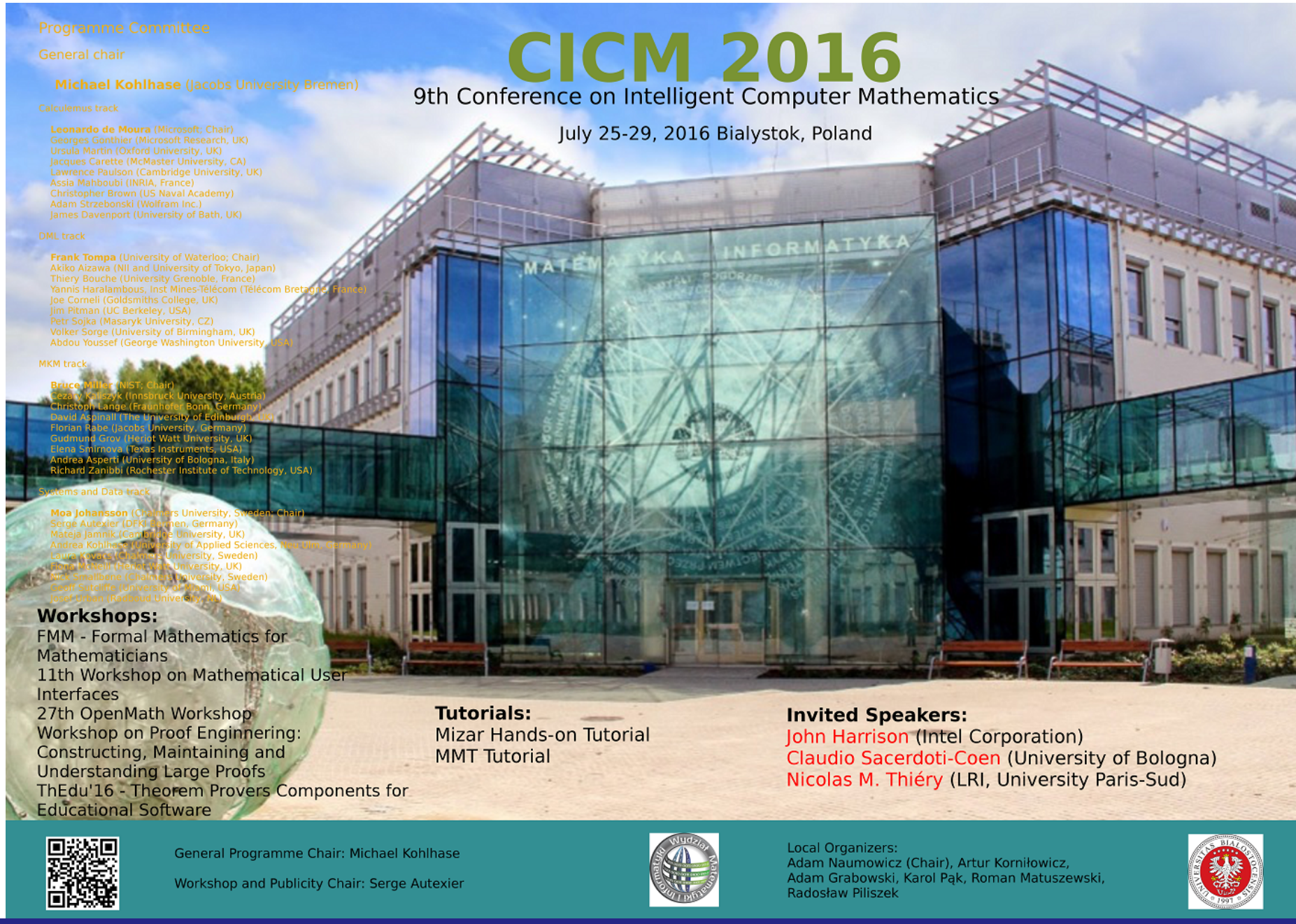
- For syntactic (identification) purposes, e.g. to force the system use one of a number of matching redefinitions, the type of a term can be explicitely qualified to one which is less specific, e.g.
  `1 qua real number`
  whereas in standard environments the constant has the type `natural number` and 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.

Types in MIZAR inference checking

- 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`.


Programme Committee
General chair
Michael Kohlhase (Jacobs University Bremen)
Calculemus track
Leonardo de Moura (Microsoft; Chair)
Georges Gonthier (Microsoft Research, UK)
Ursula Martin (Oxford University, UK)
Jacques Carette (McMaster University, CA)
Lawrence Paulson (Cambridge University, UK)
Assia Mahboubi (INRIA, France)
Christopher Brown (US Naval Academy)
Adam Strzebonski (Wolfram Inc.)
James Davenport (University of Bath, UK)
DML track
Frank Tompa (University of Waterloo; Chair)
Akiko Aizawa (NII and University of Tokyo, Japan)
Thiery Bouche (University Grenoble, France)
Yannis Haralambous, Inst Mines-Télécom (Télécom Bretagne, France)
Joe Corneli (Goldsmiths College, UK)
Jim Pitman (UC Berkeley, USA)
Petr Sojka (Masaryk University, CZ)
Volker Sorge (University of Birmingham, UK)
Abdou Youssef (George Washington University, USA)
MKM track
Bruce Miller (NIST; Chair)
Cezary Kaliszyk (Innsbruck University, Austria)
Christoph Lange (Fraunhofer, Bonn, Germany)
David Aspinall (The University of Edinburgh, UK)
Florian Rabe (Jacobs University, Germany)
Gudmund Grov (Heriot Watt University, UK)
Elena Smirnova (Texas Instruments, USA)
Andrea Asperti (University of Bologna, Italy)
Richard Zanibbi (Rochester Institute of Technology, USA)
Systems and Data track
Moa Johansson (Chalmers University, Sweden; Chair)
Serge Autexier (DFKI, Bremen, Germany)
Mateja Jamnik (Cambridge University, UK)
Andrea Kohlhase (University of Applied Sciences, Neu-Ulm, Germany)
Laura Kovacs (Chalmers University, Sweden)
Ursula Martin (Queen Mary University, UK)
Nick Smallbone (Chalmers University, Sweden)
Geoff Sutcliffe (University of Miami, USA)
Josef Urban (Radboud University, NL)
CICM 2016
9th Conference on Intelligent Computer Mathematics
July 25-29, 2016 Bialystok, Poland
MATEMATYKA
INFORMATYKA
Workshops:
FMM - Formal Mathematics for Mathematicians
11th Workshop on Mathematical User Interfaces
27th OpenMath Workshop
Workshop on Proof Enginnering: Constructing, Maintaining and Understanding Large Proofs
ThEdu'16 - Theorem Provers Components for Educational Software
Tutorials:
Mizar Hands-on Tutorial
MMT Tutorial
Invited Speakers:
John Harrison (Intel Corporation)
Claudio Sacerdoti-Coen (University of Bologna)
Nicolas M. Thiéry (LRI, University Paris-Sud)
General Programme Chair: Michael Kohlhase
Workshop and Publicity Chair: Serge Autexier
Local Organizers:
Adam Naumowicz (Chair), Artur Korniłowicz,
Adam Grabowski, Karol Pąk, Roman Matuszewski,
Radosław Piliszek