Mizar

Adam Naumowicz

Faculty of Computer Science,
University of Bialystok, Poland

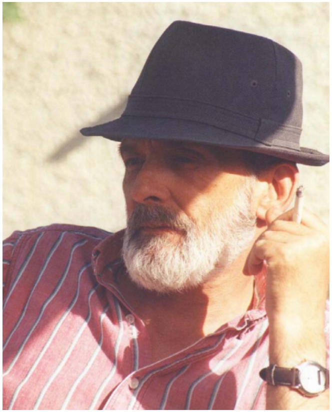
EuroProofNet School on Natural Formal Mathematics, Bonn,
June 3rd, 2025

MIZAR and (Natural) Formal Mathematics

- 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)

Andrzej Trybulec - the Creator of MIZAR

- On November 14, 1973 he made the first public presentation of MIZAR: at a seminar in the Institute of Library Science and Scientific Information at Warsaw University
- In 1975 he obtained a PhD in mathematics from Polish Academy of Sciences under Karol Borsuk. Thesis: *On some properties of the movable compacta* (topology)
- In 1977 he published the first paper description of MIZAR : Trybulec, A., Informationslogische Sprache Mizar, Dokumentation-Information, Heft 33, Ilmenau, 1977



Original Vision of MIZAR

- The papers should be stored in a computer and later, at least partially, translated into natural languages
- The papers should be formal and concise
- It would form a basis for the construction of an automated information system for mathematics
- It would facilitate detection of errors, verification of references, elimination of repeated theorems, etc.
- It would open a way to machine assisted education of the art of proving theorems
- It would enable automated generation of input into typesetting systems

Formalizing Mathematics - Mizar as a Proof Assistant

- Keeping 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
- The computer input language should facilitate various purposes of developing mathematical proofs
  - Convincing readers
  - Documenting the work
  - Presenting the results

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
    - S. Jaśkowski, On the rules of supposition in formal logic. *Studia Logica*, 1, 1934.
    - F. B. Fitch, *Symbolic Logic. An Introduction*. The Ronald Press Company, 1952.
    - K. Ono, On a practical way of describing formal deductions. *Nagoya Mathematical Journal*, 21, 1962.
- The system uses a **declarative style** of writing proofs (mostly **forward reasoning**) - resembling mathematical practice

Mizar Mathematical Library - MML

"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 - 5.94.1493
  - includes 1498 articles written by over 270 authors
  - over 73000 theorems
  - over 14000 definitions
  - over 900 schemes
  - over 19000 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 language evolves over time
  - Adding new features
  - Improving expressiveness
  - Reducing the De Bruijn factor (the loss factor computed as the size of the formal proof over the size of the informal proof)

Example from 1970s

```
BEGIN
((EX X ST (FOR Y HOLDS P2[X,Y])) > (FOR X HOLDS (EX Y ST P2[Y,X])))
  PROOF
    ASSUME THAT A: (EX X ST (FOR Y HOLDS P2[X,Y]));
    LET X BE ANYTHING;
    CONSIDER Z SUCH THAT C: (FOR Y HOLDS P2[Z,Y]) BY A;
    SET Y = Z;
    THUS D: P2[Y,X] BY C;
  END
END
```

Excerpt from 1970s IBM 370 Pascal P8000 Implementation

```
FOR J := 1 TO LSNBR-1 DO
     BEGIN LSNT := SNTARRAYOJE;
       IF (EVALOLSNBRE<>EVALOJE) AND (SNTSORT=LSNTa.SNTSORT) THEN
         CASE SNTSORT OF
           REDUCED,UNIVERSAL:;
           ATOMIC:
             IF PRED = LSNTa.PRED THEN
               PREPARELIST(ATOMARGS,LSNTa.ATOMARGS);
           EQUALITY:
             BEGIN PREPARE(LEFT,LSNTa.LEFT);
               PREPARE(RIGHT,LSNTa.RIGHT);
               PREPARE(LEFT,LSNTa.RIGHT);
               PREPARE(RIGHT,LSNTa.LEFT)
             END
         END
       END;
```

The Mizar Language - ctd. (1)

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

$\neg\alpha$	`not` α
$\alpha \wedge \beta$	α `&` β
$\alpha \vee \beta$	α `or` β
$\alpha \rightarrow \beta$	α `implies` β
$\alpha \leftrightarrow \beta$	α `iff` β
$\exists_x \alpha$	`ex x st` α
$\forall_x \alpha$	`for x holds` α
$\forall_{x:\alpha} \beta$	`for x st` α `holds` β

The MIZAR Language - ctd. (2)

- Each quantified variable has to be given its **type**, so the quantifiers actually take the form

  ```
  for x being set holds ...
  ```

  or

  ```
  ex y being real number st ...
  ```

  where `set` and `real number` represent examples of types
- MIZAR allows to globally assign this type to selected variable names with a *reservation*

  ```
  reserve x,y 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 Mizar Language - ctd. (3)

- The formulas

  ```
  for x holds for y holds ...
  ```

  or

  ```
  for x holds ex y st ...
  ```

  may be shortened to

  ```
  for x for y holds ...
  ```

  and

  ```
  for x ex y st ...
  ```

- Instead of writing

  ```
  for x holds for y holds ...
  ```

  or

  ```
  ex x st ex y st ...
  ```

  more convenient forms with lists of variables are allowed

  ```
  for x,y holds ...
  ```

  and

  ```
  ex x,y 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	aggregate	all	and	antonym	are	as
associativity	assume	asymmetry	attr	axiom	be	begin
being	by	canceled	case	cases	cluster	coherence
commutativity	compatibility	connectedness	consider	consistency	constructors	contradiction
correctness	def	deffunc	define	definition	definitions	defpred
do	does	end	environ	equalities	equals	ex
exactly	existence	expansions	for	from	func	given
hence	hereby	holds	idempotence	identify	if	iff
implies	involutiveness	irreflexivity	is	it	let	means
mode	non	not	notation	notations	now	of
or	otherwise	over	per	pred	prefix	projectivity
proof	provided	qua	reconsider	redefine	reduce	reducibility
reflexivity	registration	registrations	requirements	reserve	sch	scheme
schemes	section	selector	set	sethood	st	struct
such	suppose	symmetry	synonym	take	that	the
then	theorem	theorems	thesis	thus	to	transitivity
uniqueness	vocabularies	when	where	with	wrt	

The Mizar Language - ctd. (5)

- 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-`)

Types of Mizar Objects

- 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 successful 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;
```

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`

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)

Encoding Proofs in MIZAR

- For any formula Φ its proof may take the form of a proof block in which the same formula is finally stated as a conclusion after the **thus** keyword

```
Φ
proof
...
 thus Φ;
end;
```

Encoding Proofs in Mizar - ctd. (2)

- If the formula to be proved is a conjunction, then the proof should contain two conclusions:

```
Φ1 & Φ2
proof
...
 thus Φ1;
...
 thus Φ2;
end;
```

Encoding Proofs in MIZAR - ctd. (3)

- When proving an implication, the most natural proof is the one where we first assume the antecedent and conclude with the consequent:

```
Φ₁ implies Φ₂
proof
 assume Φ₁;
...
 thus Φ₂;
end;
```

Encoding Proofs in MIZAR - ctd. (4)

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

```
Φ1 iff Φ2
proof
...
 thus Φ1 implies Φ2;
...
 thus Φ2 implies Φ1;
end;
```

Encoding Proofs in MIZAR - ctd. (5)

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

```
Φ₁ iff Φ₂
proof
 hereby
  assume Φ₁;
...
  thus Φ₂;
 end;
 assume Φ₂;
...
 thus Φ₁;
end;
```

Encoding Proofs in MIZAR - ctd. (6)

- Disjunction is usually proved by assuming that the first disjunct does not hold and then to proving the other:

```
Φ₁ or Φ₂
proof
 assume not Φ₁;
...
 thus Φ₂;
end;
```

Encoding Proofs in MIZAR - ctd. (7)

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

```
Φ
proof
 assume not Φ;
...
 thus contradiction;
end;
```

Encoding Proofs in MIZAR - ctd. (8)

- 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;
```

Encoding Proofs in MIZAR - ctd. (9)

- A proof of an existential statement must provide a witness term *a* and an appropriate conclusion

```
ex a being Θ st Φ
proof
...
 take a;
...
 thus Φ;
end;
```

Getting the Proof Structure Correct

- 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
  - α `implies` β is changed into `not(`α `&` `not` β`)`
  - α `iff` β is changed into α `implies` β `&` β `implies` α, i.e. `not(`α `&` `not` β`)` `&` `not(`β `&` `not` α`)`
  - conjunction is associative but not commutative

Justifications in Mizar

- 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

$$\frac{\alpha^1, \ldots, \alpha^k, \neg\beta}{\perp}$$

Justifications in MIZAR - ctd. (1)

- A disjunctive normal form (DNF) of the premises is then created and the system tries to refute it

$$\frac{([\neg]\alpha^{1,1} \wedge \cdots \wedge [\neg]\alpha^{1,k_1}) \vee \cdots \vee ([\neg]\alpha^{n,1} \wedge \cdots \wedge [\neg]\alpha^{n,k_n})}{\perp}$$

where $\alpha^{i,j}$ are atomic or universal sentences (negated or not)

Justifications in Mizar - ctd. (2)

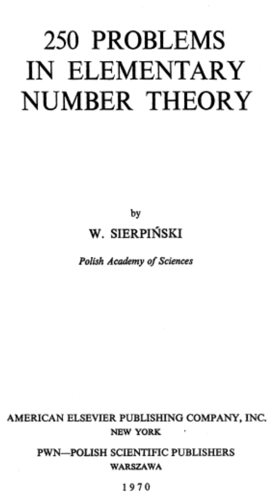
- For the inference to be accepted, all disjuncts must be refuted; so in fact *n* inferences are checked

$$\frac{[\neg]\alpha^{1,1} \wedge \cdots \wedge [\neg]\alpha^{1,k_1}}{\perp}$$

$$\ldots$$

$$\frac{[\neg]\alpha^{n,1} \wedge \cdots \wedge [\neg]\alpha^{n,k_n}}{\perp}$$

Task for the Hands-on Session



250 PROBLEMS
IN ELEMENTARY
NUMBER THEORY

by

W. SIERPIŃSKI

Polish Academy of Sciences

AMERICAN ELSEVIER PUBLISHING COMPANY, INC.
NEW YORK

PWN—POLISH SCIENTIFIC PUBLISHERS
WARSZAWA

1970

Statement - Informal and Formal

As stated in Sierpiński's book:

101. Show that the assertion that by changing only one decimal digit one can obtain a prime out of every positive integer is false.

A possible formulation in MIZAR :

```
theorem
  not for n being positive Nat
  ex i,j being Element of NAT st 0 <= j <= 9 &
  value(Replace(digits(n,10),i,j),10) is prime;
```

Running the System

- Pre-compiled distributions are available for Windows, Linux and MacOs
- Interfaces: CLI, Emacs MIZAR Mode by Josef Urban, VS Code Extension by H. Taniguchi and K. Nakasho
    - 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

Internal Processing Modules

- 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

Enhancing Mizar Texts

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

Importing Notions from the Mizar 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)

Recommended Further Reading about MIZAR

- Grzegorz Bancerek et al., Mizar: State-of-the-Art and Beyond. CICM 2015, LNAI 9150, pp. 261-279, 2015
- A. Grabowski, A. Kornilowicz and A. Naumowicz, Mizar in a Nutshell, Journal of Formalized Reasoning 3(2), pp. 153-245, 2010. (https://jfr.unibo.it/article/view/1980/1356)
- A. Trybulec, Checker (a collection of e-mails compiled by F. Wiedijk). (https://www.cs.ru.nl/~freek/mizar/by.pdf)
- F. Wiedijk, Mizar: An Impression. (https://www.cs.ru.nl/~freek/mizar/mizarintro.pdf)
- F. Wiedijk, Writing a Mizar article in nine easy steps. (https://www.cs.ru.nl/~freek/mizar/mizman.pd)
- F. Wiedijk (ed.), The Seventeen Provers of the World. LNAI 3600, Springer Verlag 2006. (https://www.cs.ru.nl/~freek/comparison/comparison.pdf)
- Markus Wenzel and Free Wiedijk. A comparison of the mathematical proof languages Mizar and Isar. In Journal of Automated Reasoning, 29, 2002. (https://www.cs.ru.nl/~freek/pubs/romantic.pdf)