Mizar in a Nutshell

ADAM GRABOWSKI
Institute of Mathematics, University of Białystok
and
ARTUR KORNIŁOWICZ and ADAM NAUMOWICZ
Institute of Informatics, University of Białystok

This paper is intended to be a practical reference manual for basic Mizar terminology which may be helpful to get started using the system. The paper describes most important aspects of the Mizar language as well as some features of the verification software.

1. INTRODUCTION

Mizar is the name of a formal language designed by Andrzej Trybulec for writing strictly formalized mathematical definitions and proofs, but is also used as the name of a computer program which is able to check proofs written in this language. The Mizar project encompasses the development of both these aspects. The combination of a convenient language with efficient software implementation made Mizar successful as a proof assistant with numerous applications and a powerful didactic tool. Currently the main effort of the Mizar community is to build the Mizar Mathematical Library (MML), the comprehensive repository of interrelated definitions and proved theorems which can be referenced and used in new articles.

This paper is not a step by step tutorial of Mizar, but rather a reference manual describing the expressiveness of its language and the capabilities of the verification system. The main purpose is to help the readers get acquainted with basic Mizar terminology and jargon. After studying the paper, the readers should be ready to understand all Mizar texts available in MML and start individual experiments with writing and verifying their own proofs using Mizar.

2. LANGUAGE

The Mizar language is designed to be as close as possible to the language used in mathematical papers and at the same time to be automatically verifiable. The two goals are achieved by selecting a set of English words and phrases which occur most often in informal mathematics. In fact, Mizar is intended to be close to the mathematical vernacular on the semantic level even more than on the level of the actual grammar. Therefore the syntax of Mizar is much simplified compared to the natural language, stylistic variants are not distinguished and instead of English words in some cases their abbreviations are used.

In particular, the language includes the standard set of first order logical connectives and quantifiers for forming formulas and also provides means for using free second order variables for forming schemes of theorems (infinite families of theorems, e.g. the induction scheme, see Section 3.5.1). The rest of Mizar syntactic

Journal of Formalized Reasoning Vol. 3, No. 2, 2010 Pages 153–245.

constructs is used for writing proofs and defining new mathematical objects. A text written in that language is usually referred to as an *article*.
Every article is a plain ASCII text file with lines not longer than 80 characters, starting with keywords **environ** and **begin**. The proper article is what follows the keyword **begin**. Sandwiched between these two keywords one can specify one or more of eight types of directives telling the system what notions from what pre-existing MML articles will be needed in the proper article (see Section 4.4), and possibly the name of external files containing the symbols of new entities the user will want to define in the proper article.
The complete grammar of a Mizar article is presented in Appendix B.

The table below lists all Mizar reserved words (please mind that the language is case-sensitive):

according	aggregate	and	antonym
as	associativity	assume	asymmetry
attr	be	begin	being
by	canceled	case	cases
cluster	coherence	commutativity	compatibility
connectedness	consider	consistency	constructors
contradiction	correctness	def	deffunc
define	definition	definitions	defpred
end	environ	equals	ex
exactly	existence	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	@proof
provided	qua	reconsider	redefine
reflexivity	registration	registrations	requirements
reserve	sch	scheme	schemes
section	selector	set	st
struct	such	suppose	symmetry
synonym	take	that	the
then	theorem	theorems	thesis
thus	to	transitivity	uniqueness
vocabularies	when	where	with
wrt			

Please note that some of the words have different meaning depending on the context, e.g. `for` (compare its usage in Sections 2.1 and 2.3.6), `set` (compare its usage in Sections 2.2.3 and 2.3.3), or `the` (compare of the usage of it in Sections 2.3.3 and the two distinct usages of it in 2.3.5). There are also synonyms `be` and `being`, while `hereby` can be treated as a shorthand for `thus now`. The words `according`, `aggregate`, `associativity`, `exactly`, `prefix`, `section`, `selector`,

`to`, `transitivity` and `wrt` are reserved, but currently not implemented in the processing software. The language also utilizes a set of special symbols:

```
,  ;  :  (  )  [  ]  {  }  =  &  ->
.=  ...  $1  $2  $3  $4  $5  $6  $7  $8  $9  $10  (#  #)
```

One also has to keep in mind that an occurrence of a double colon (`::`) in Mizar texts starts a one-line comment. More information on user-defined identifiers, symbols that may be used for introduced notions and special processing of numerals can be found in Section 3.1 describing the Mizar lexical analyzer.

2.1 Formulas

Mizar is essentially a first order system, so the statements one can write and check for correctness with the Mizar verifier are composed of atomic (predicative) formulas combined with classical logic connectives and quantifiers. However, Mizar maintains the type information associated with all terms, so the set of atomic formulas contains also qualified and attributive formulas (see Section 2.3 what kinds of notions can be defined in Mizar).

The table below shows the Mizar representation of standard logical connectives and quantifiers:

$\neg\alpha$	`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

In fact, 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. If several statements are to be written with the same type of variables, Mizar allows to globally assign this type to selected variable names with a *reservation*, e.g.:

```
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.

Also for the users' convenience, writing

```
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 ...
```

respectively. Moreover, instead of writing e.g.:

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

or

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

more convenient forms with lists of variables are allowed, namely

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

and

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

.

Let us also mention here that the binding force of quantifiers is weaker than that of connectives.

2.2 Proofs

Just like standard mathematical papers, Mizar articles consist of introduced definitions and statements about them, with proofs being the biggest part of the text. Therefore let us first focus on proofs.

By its design, Mizar supports writing proofs in a declarative way (i.e. mostly forward reasoning), resembling the standard mathematical practice. The proofs written in Mizar are constructed according to the rules of the Jaśkowski style of natural deduction [6], or similar systems developed independently by F.B. Fitch [3] or K. Ono [15]. It is this part of the Mizar language that has had the biggest influence on other systems and became the inspiration to develop similar proof layers on top of several procedural systems. To name the most important ones, there was the system Declare by D. Syme [18], the Mizar mode for HOL by J. Harrison [5], the Isar language for Isabelle by M. Wenzel [22], Mizar-light for HOL-light by F. Wiedijk [24] and most recently the declarative proof language (DPL) for Coq by P. Corbineau [2]. The Mizar way of writing proofs was also the model for the notion of 'formal proof sketches' developed by F. Wiedijk [23]. Below we present the typical *proof skeletons* reflecting the structure of the statement being proved. However, most proofs can be stated in many ways, because all Mizar formulas are internally expressed in a simplified "canonical" form (see Section 3.2 for the description of Mizar *semantic correlates*). As far as the proof structure agrees with the semantic correlate of the proved formula, it is valid. The correctness of a given proof skeleton is checked by the Mizar verifier's module called Reasoner (see Section 3.4).

2.2.1 *Proof skeletons.* 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. In practice, this only makes sense for atomic formulas:

```
Φ
proof
  ...
 thus Φ;
end;
```

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

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

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;
```

Mizar interprets an equivalence statement 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;
```

In this case, the level of proof nesting can be slightly reduced if we use the following skeleton:

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

The typical way to prove a disjunction is to assume that the first disjunct does not hold and then to prove the other:

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

```
 thus Φ2;
end;
```

Although from the logical point of view we could as well assume that the second disjunct does not hold and complete the proof by showing the validity of the first disjunct, the users should be warned that this proof skeleton is invalid, because at the level of semantic correlates used by the REASONER the conjunction is not commutative (see 3.4).

Also note that any formula can be proved using the *reductio ad absurdum* method, like in this indirect proof:

```
Φ
proof
 assume not Φ;
  ...
 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;
```

Please note that if a has a reserved type Θ, the '**be** Θ' and '**being** Θ' phrases are not necessary and can be omitted. 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;
```

Obviously the type of a must fit (in the Mizar jargon: *widen to*) the one used in the quantifier (see Section 2.3.3 for the discussion of the Mizar type system and the *widening* relation).

2.2.2 *Justification.* 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 (see Section 3.5). If a statement produces an inference which is obvious to the CHECKER, it suffices to end it with a semicolon (;) and the CHECKER will not report any errors. In most cases, however, the lack of any justification is reported by inserting the following error flag in the Mizar text:

```
::>                                  *4
::> 4: This inference is not accepted
```

indicating with the `*4` marker the position of the missing justification in a previous line. The CHECKER keeps producing the error message until the user justifies the inference. As presented above, the justification may take the form of a proof block that starts with the keyword `proof` and finishes with `end` and a closing semicolon. When a given statement can be found obvious by the CHECKER as a consequence of previously stated statements, the proof can be reduced to a *straightforward justification* that consists of the `by` keyword followed by a comma-separated list of references to appropriate statements, terminated with a semicolon. The references can point to statements previously marked with respective labels, e.g.

```
lb1: Φ1;
...
lb2: Φ2;
...
Φ by lb1,lb2;
```

As the proofs can be nested, labels can be used to mark formulas on various proof levels overriding any previously defined ones in a given proof block.

One may also use references to theorems from separate Mizar articles (local or available in the MML). Then the reference takes the form `<FILENAME>:<number>` or `<FILENAME>:def <number>` where `<FILENAME>` is the name of the referenced article and `<number>` corresponds to a certain theorem or definition in that article. See also Section 3.1 for the definition of valid label and file identifiers.

Please also note that it is very typical for proof steps to form linear reasoning paths. To facilitate this, instead of

```
lb1: Φ1;
...
lb2: Φ2;
Φ by lb1,lb2;
```

one can write the statements linked with **then**:

```
lb1: Φ1;
...
Φ2;
then Φ by lb1;
```

In a similar way, linear reasoning that leads to a conclusion may be closed with the **hence** keyword replacing **thus** and **then**.

If a statement should be justified using a scheme (a statement with free second order variables), the simple justification utilizes the **from** keyword instead of **by**, indicating to the Mizar verifier that a different module, the SCHEMATIZER, should be used to check the correctness of that inference. In that case the reference takes the form `<FILENAME>:sch <number>` for a scheme with a certain number contained in the article `<FILENAME>` or is an identifier if the reference is made to a local scheme. As the schemes usually have their premises, before the terminating semicolon a complete scheme reference requires also a comma-separated list of references to

formulas to be substituted for scheme premises put in a bracket. Please note that unlike the references in a straightforward justification after **by**, where the order in which labels are stated should not be relevant, in the case of scheme references they must match exactly the order of the scheme's premises. See Section 3.5.1 for more information on schemes in Mizar with some examples.

2.2.3 *Auxiliary proof elements.* Having introduced the basic structure of Mizar proofs and how statements can be justified, in order to fully understand any Mizar proof, one also has to know the semantics of other Mizar keywords that may constitute proofs. Below we briefly present these auxiliary proof elements and show examples of their usage.

Let us start with two constructions which are not really necessary, but their use approximates mathematical vernacular better. One of them is *iterative equality* of the form

$$\begin{array}{lll} \alpha_1 & \texttt{=} & \alpha_2 \ \textit{justification} \\ & \texttt{.=} & \alpha_3 \ \textit{justification} \\ & \ldots & \\ & \texttt{.=} & \alpha_n \ \textit{justification}; \end{array}$$

where α_i are terms, which can be transformed into the form without `.=`:

$$\begin{array}{lll} \alpha_1 & \texttt{=} & \alpha_2 \ \textit{justification}; \textbf{ then} \\ \alpha_1 & \texttt{=} & \alpha_3 \ \textit{justification}; \textbf{ then} \\ & \ldots & \\ \alpha_1 & \texttt{=} & \alpha_n \ \textit{justification}; \end{array}$$

The last formula in the above sequence is equivalent to the whole iterative equality.

The other construct is the *Fraenkel operator,* i.e. a version of set comprehension in ZF. For proper identification of Fraenkel terms, `SUBSET` should be added to the **requirements** environment directive.

The general form of the Fraenkel operator is as follows:

$$\{t \text{ where } v_1 \text{ is } \Theta_1, \ldots, v_n \text{ is } \Theta_n : \Phi\}$$

where t is a term, v_i is a variable of the type Θ_i. For every i the type Θ_i has to widen to the type `Element of A`, where A is arbitrary, not necessarily non-empty, set, and Φ is a formula. If not, the error `*129` is reported. Listed variables do not need occur in t and Φ.

The most common form of unfolding this term is given below:

```
    x in { t where t is Element of A : not contradiction }; then
    consider t1 being Element of A such that
A1: t1 = x;
```

set can be used to introduce abbreviations for more complex terms for easier typing:

```
set A = X \/ (Y /\ Z);
set B = (X \/ Y) /\ (X \/ Z);
A = B by XBOOLE_1:24;
```

and A and B can be further used in the rest of the proof.

reconsider forces the system to treat a given term as if its type was the one stated; 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.

```
let k, n be Nat;
assume k >= n; then
reconsider m = k - n as Element of NAT by NAT_1:21;
```

consider introduces a new constant of a given type satisfying stated conditions.

```
  consider k being Integer such that
A1: m = 2 * k + 1;
```

It is also the method of accessing variables under an existential quantifier:

```
  ex k being Integer st m = 2 * k + 1 & k > 0; then
  consider k1 being Integer such that
A1: m = 2 * k1 + 1 & k1 > 0;
```

Observe that the ordering of conjuncts matters with this construct; if we try to swap the conjunction under `A1` label, we obtain `*4` error.

given is an abbreviation for an assumption (**assume**) of an existential statement (**ex**) followed by **consider**, so that instead of typing

```
  assume ex n being Integer st x = 2 * n; then
  consider n being Integer such that
A1: x = 2 * n;
```

it is shorter to write

```
  given n being Integer such that
A1: x = 2 * n;
```

thesis is a keyword playing a role of the phrase *which has to be proved*; so it can significantly shorten the last statement of a proof replacing the dynamically changing thesis; it is only allowed within a proof, otherwise the error `*65` is reported.

contradiction can be read logically as *falsum*. It is often used in the negated context; as **not contradiction** occurs very often in Fraenkel operators. In proofs by contradiction it plays the role of the thesis, hence **thesis** and **contradiction** can be exchanged in such cases.

per cases facilitates the method of proving by exhaustion. If the list of cases is exhaustive based on a logical tautology (usually the law of excluded middle), it does not need its own justification, otherwise it should be justified by a list of references (a nested proof is not permitted here). This construct does not need a closing **end**.

suppose is a word starting new case in the proof by exhaustion (**per cases**). The thesis remains unchanged, i.e. is identical under every case. Below we present a simple proof with two cases:

```
reserve A,B,C for set;
```

```
theorem
  A \ B c= C implies A c= B \/ C
  proof
    assume
A1: A \ B c= C;
    let x be set;
    assume
A2: x in A;
    per cases;
    suppose x in B;
      hence thesis by XBOOLE_0:def 3;
    end;
    suppose not x in B;
      then x in A \ B by A2,XBOOLE_0:def 5;
      then x in C by A1,TARSKI:def 3;
      hence thesis by XBOOLE_0:def 3;
    end;
  end;
```

Please note that the **theorem** keyword makes the statement exportable (it is then possible to refer to it in other articles).

case is similar to **suppose**; although if the latter faithfully reflects the idea of proof by cases, **case** can be used, e.g. to prove a disjunction of conjuncts. Both **suppose** and **case** have to be bracketed with their closing **end**.

now opens diffuse reasoning (that requires a closing **end**). The proved statement is not written explicitly, letting the Reasoner to reconstruct it dynamically – useful in the case of a long list of generalized variables and a long thesis.

A general form is

```
now
  ...
  thus Alpha(x);
end;
```

where dots replace reasoning (sequence of formulas) and it is just a proof of `Alpha(x)`. Conversion of arbitrary diffuse statement into formula is usually not problematic, maybe with the exception of

```
now
  assume Alpha;
  ...
  take x;
  thus Beta;
end;
```

where exemplification can be translated back into the formula

```
  Alpha implies ex x st Beta;
```

hereby can be treated as a kind of shorthand for **thus** and **now**. This construction allows to omit one nesting of the proof:

```
hereby
  assume k in n;
::here the proof...
  hence k < n;
end;

k in n implies k < n
proof
  assume k in n;
::here the proof...
  hence k < n;
end;
```

2.3 Defining notions

In this section we will show how to define new notions in the Mizar language. It is important to know what kinds of notions can be introduced and how they correspond to notions used in standard 'pen-and-paper' mathematics.

To start, let us consider the following formulas:

$$\forall_{n\in\mathbb{N}}\ n+1>0$$

the sum of a natural number and 1 is positive

for n being a natural number it holds that $n+1>0$

for n being a natural number it holds that $n+1$ is positive

All the above formulas express the same property, but use different syntactic and logical devices. First of all, there is a universal quantifier, spelled as $\forall$ or "for ... holds ...". There is a variable n, numerals 0 and 1, which construct a *term* using the $+$ *operation symbol*. Whenever n is used, we always state what sort of values it can take, in other words what is the *type* of it ('natural number' in this case). The type may consist of a single noun ('number'), or may be made more precise by prepending some adjectives (like 'natural'), so that the statement is true. In the above formulas we also have a relation symbol $>$, which is used to create a predicative formula, and also an attributive formula created using the adjective 'positive'.

All these constructs are supported in Mizar. Namely, *predicates* are used to construct formulas, *modes* are constructors of types, adjectives are constructed with *attributes*, and *functors* are constructors of terms.

All new notions are introduced within a *definitional block* which starts with the **definition** keyword and finishes with a corresponding **end;**. Within such a block there may appear:

—arguments (if required) with their corresponding types, called *loci* (*locus* in singular) introduced with the **let** keyword

—possibly some assumptions for the definition that follow **assume** or **given**

—the main part of the definition – a detailed description of this part will follow in next sections

—correctness conditions if they are needed to guarantee the soundness of the definition
—properties of the introduced notion (e.g. commutativity of an operation, or reflexivity of a relation), if applicable
—additionally, a definitional block may contain local reasonings that may be used in several proofs within the current block.

A single definitional block may be used to introduce many notions. However, a new notion becomes *available* only after the block is closed with **end;**. For example, in the following text, the Mizar system reports an error marked with the `*102` flag indicating an unknown predicate:

```
definition
  pred pred_symbol_1 means not contradiction;
  pred pred_symbol_2 means pred_symbol_1;
::>                                    *102
::> 102: Unknown predicate
end;
```

To fix this problem, one should introduce the latter predicate in a separate definitional block:

```
definition
  pred pred_symbol_2 means pred_symbol_1;
end;
```

Now let us concentrate on the loci declarations. First of all, none of the arguments may be omitted in the declaration part of a definitional block. For example, when defining a subset's complement within a superset, both arguments must be declared in the loci section that is opened with the **let** keyword:

```
definition
  let E be set, A be Subset of E;
  func A‘ -> Subset of E equals
  E \ A;
  ...
end;
```

The following definition is, therefore, incorrect:

```
definition
  let A be Subset of E;
::>                  *140
  func A‘ -> Subset of E equals
  ...
end;
::>,140
::> 140: Unknown variable
```

Even if the variable `E` has already been locally defined in the article, an error is reported:

```
set E = the set;
...
definition
  let A be Subset of E;
::>             *222
  func A‘ -> Subset of E equals
  ...
end;
::> 222: Local constants are not allowed in library items
```

Although taking the complement is in fact a two-argument operation, defining it as above yields one *hidden argument* (`E`) and one *visible argument* (`A`). An alternative definition might look like this:

```
definition
  let E,A;
  func ‘(E,A) -> Subset of E equals
  E \ A;
  ...
end;
```

where both arguments are visible. But, as in standard mathematics practice, such definitions are not used frequently. The Mizar system, however, does not require that the minimal number of visible arguments be used. Yet the arguments cannot be repeated:

```
definition
  let E,A;
  func ‘(E,A,A) -> Subset of E equals
::>          *141
  E \ A;
  ...
end;
::> 141: Locus repeated
```

The key property of Mizar loci is that every locus must be used as a parameter of another locus, or a visible argument of the defined notion. In our example definition, the set `E` is a parameter of the type of the its subset `A`, so it does not need to be a visible argument. However, the following definition would be incorrect:

```
definition
  let E be set, A be Subset of E;
  func ‘E -> Subset of E equals
::>    *100
  E \ A;
  ...
::> 100: Unused locus
```

because `A` does not comply with that requirement.

Throughout this paper, for reader's convenience, in contexts where in the description of defined notions it is not necessary to provide their full meaning, it may

be enough to provide the symbol of the defined notion together with the number of its left and right arguments, called its *format*. If we add to the format also the information on the types of all loci (and the result type if applicable), then we get the definition's *pattern*.

With that general terminology set, let us move on to the aspects specific to the definitions of certain kinds of constructors.

2.3.1 *Predicates.* The definitions of predicates, just like all other constructors, are stated in a definitional block. After the list of loci declarations, the **pred** keyword is followed by the defined predicate's *format*. The arity is specified with a certain number of loci symbols on both sides of the predicate symbol, e.g.:

```
definition
  let f,g be Function; let A be set;
  pred f,g equal_outside A means
:Def:
  f|(dom f \ A) = g|(dom g \ A);
end;
```

Here the `equal_outside` predicate has three arguments: two left arguments and one right argument. Then, after the **means** keyword follows a label (optional) which may be later used to refer to that definition and the *definiens*, i.e. the formula describing the defined relation, in our example: `f|(dom f \ A) = g|(dom g \ A)`.

Referring to a definition means making a reference to its corresponding *definitional theorem* generated implicitly by the system, in our example it has the following meaning:

```
for f,g being Function, A being set holds
f,g equal_outside A iff f|(dom f \ A) = g|(dom g \ A);
```

The Mizar language supports also "piecewise definitions", where the actual meaning is different in different setting e.g.:

```
definition
  let x,y be ext-real number;
  pred x <= y means
  ex p,q being Element of REAL st p = x & q = y & p <= q
    if x in REAL & y in REAL
  otherwise x = -infty or y = +infty;
end;
```

In that case, the definitional theorem has the form:

```
for x,y being ext-real number holds
(x in REAL & y in REAL implies
  (x <= y iff ex p,q being Element of REAL st
              p = x & q = y & p <= q)) &
(not x in REAL or not y in REAL implies
  (x <= y iff x = -infty or y = +infty));
```

Some of the defined notions can be accompanied in their definitional block by statements and proofs of specific *properties*. For example, a relation defined in such

a way that it is always symmetric, may have this fact recorded with the **symmetry** property within its definition. Then the Mizar verifier can use this property as an extra heuristic whenever the notion is used. More information on the properties supported by Mizar can be found in Section 2.5.

Let us also mention that any Mizar definition may be formulated under some assumptions that follow the **assume** keyword. Typically, assumptions are needed to assure that the introduced notion is well-defined, i.e. the required correctness conditions can be proved for a given definiens. Although it is not disallowed by the language, it makes little sense to use assumptions in definitions of predicates, because in that case no correctness conditions are required, so an assumption would result in an unnecessarily restricted definition.

2.3.2 *Attributes. Attributes* are constructors of adjectives. One can define an attribute with the **attr** keyword, followed by its *subject*, **is** keyword, the attribute's symbol, and the **means** keyword followed by the definiens (there can also be a label before the definiens to allow making references). It is required that the locus which plays the role of the subject be the last one in the list of all loci. Otherwise, there would be unused loci in the definition.

```
definition
  let E be set, A be Subset of E;
  attr A is proper means :Def:
  A <> E;
end;
```

The corresponding definitional theorem which is used by the CHECKER whenever a reference is made to the definition, can be formulated as follows:

```
for E being set, A being Subset of E holds
A is proper iff A <> E;
```

An important extension of the Mizar attribute system is the support for *attributes with visible arguments*. This mechanism, for example, allows to introduce the notion of `n-dimensional space` or `r,c-sized matrix`. In that case, after the **is** keyword, but before the attribute symbol the syntax allows for a list of arguments, e.g.:

```
definition
  let N be Cardinal, X be set;
  attr X is N-element means
  card X = N;
end;
```

Just like in the case of attributes without visible arguments, the system requires that the subject be specified as the last locus. In the above example, although all loci (`X` and `N`) are used in the pattern, the position of the subject (`X`) is determined. Therefore the following definition would be incorrect:

```
definition
  let X be set, N be Cardinal;
  attr X is N-element means
::>    *374
```

```
  card X = N;
end;
::> 374: Incorrect order of arguments in an attribute definition
```

2.3.3 *Modes.* Types in Mizar are constructed using *modes*. Mizar supports two kinds of mode definitions:

—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*,

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

Expandable modes are introduced with the **mode** keyword, the mode's symbol with an optional list of arguments (the arguments follow the **of** keyword), followed by the **is** keyword, a list of adjectives and the radix type. For example:

```
definition
  mode Function is Function-like Relation;
end;
```

As the name indicates, expandable modes are just shortcuts, or macros, which are internally expanded by the system into a collection of adjectives associated with a radix type. Therefore, the verifier does not need any reference to accept that an object of type `Function` has also the type `Function-like Relation`, and the other way round.

A definition of a non-expandable mode has a slightly different syntax, because the underlying property is stated explicitly in the definition. So, after the **mode** keyword, the symbol and the potential list of its arguments, there is an arrow `->` and the *mother type* followed by the **means** keyword and the definiens (which may be labeled). For example:

```
definition
  let X be set;
  mode a_partition of X -> Subset-Family of X means :Def:
  union it = X &
  for A being Subset of X st A in it holds A <> {} &
    for B being Subset of X st B in it holds A = B or A misses B;
end;
```

Please note that the **it** keyword in the definiens is used to refer to an object possessing the type being defined.

One of the key features of the Mizar type system is that the introduced types must be non-empty, i.e. there must exist at least one object of a given type. This restriction is introduced to guarantee that the formalized theory always has some denotation. Therefore mode definitions require a proof of that fact stated in the form of the **existence** condition. More information on correctness conditions can be found in Appendix A. The equivalent existence condition for expandable modes is realized by existential registrations of clusters of adjectives. So before the expression `Function-like Relation` could be used to define a new expandable mode, it must be first registered (see Section 2.6).

Let us note that the non-emptiness of types allows to introduce into reasonings variables of any type with the **consider** statement, because their existence is guaranteed. Moreover, Mizar can internally support *global choice* in the sense that it generates one object for every type, not specifying its structure,
e.g.: `the real number` is a fixed real number. Whenever the term `the real number` is used, it is always the same number, although its value is not specified.

The mother type in a mode definition is used to specify the direct predecessor of the defined mode in the tree of Mizar types (representing the type *widening* relation). The type constructed with the new mode *widens to* its mother type. The widening relation also takes into account the adjectives that come with types, i.e. the type with a shorter list of (comparable) adjectives is considered to be *wider*. The built-in type `set` is always the widest.

With the above example definition, the system automatically accepts as true that every partition is a family of subsets of the underlying set. It means that all notions defined for families of subsets can be also applied to partitions. However, to state that a certain family of subsets is in fact a partition one needs to make a reference to the corresponding definitional theorem. The theorem generated by the system for the above definition has the following meaning:

```
for X being set, IT being Subset-Family of X holds
IT is a_partition of X iff
union IT = X &
for A being Subset of X st A in IT holds A <> {} &
  for B being Subset of X st B in IT holds A = B or A misses B;
```

Please note that non-expandable modes can be redefined (see Section 2.4), while expandable modes cannot, because of their macro-like nature.

2.3.4 *Functors.* The constructors of terms in Mizar are called *functors*. A functor's definition starts with loci declarations and assumptions (if needed), just like in the case of other previously defined constructors.

Then after the **func** keyword the functor's format is specified. Mizar supports prefix, infix and suffix formats, as well as circumfix (bracket-like) formats that are useful for encoding notions like intervals, etc.

Please note that if the number of arguments on the left or right side of the functor's symbol is greater than one, then these arguments must be put in brackets.

After the format, there is an arrow `->` followed by the type of the defined operation. In case the type is **set** (the most general type that all objects automatically have), this phrase can be omitted. Finally, the meaning of the term is provided after the **means** keyword followed by the (usually labeled) definiens, e.g.:

```
definition
  let X,Y be set;
  func X \/ Y -> set means
  x in it iff x in X or x in Y;
  existence;
  uniqueness;
end;
```

Within the definiens, the keyword `it` may be used for referring to the term being defined, with the declared type.

A functor is well-defined if its definition contains proofs of two correctness conditions: **existence** and **uniqueness**. Please see Appendix A for more information on the formulas to be proved as correctness conditions.

As in the case of other definitions, a reference made to a definition is in fact a reference to its definitional theorem. In our example it would look like this:

```
for X,Y,Z being set holds Z = X \/ Y iff
for x being set holds x in Z iff x in X or x in Y;
```

It is often the case that the definiens of a functor has the form `it = ...`. In other words, the object being defined is simply equal to some term which may be constructed. Then it is advisable to use a slightly different syntax to define such functors, where instead of the **means** keyword and the standard definiens one should use **equals** followed by the term to which the new object should be equal. For example, symmetric difference can be defined as below:

```
definition
  let X, Y be set;
  func X \+\ Y -> set equals
  (X \ Y) \/ (Y \ X);
  coherence;
end;
```

In that case, the required correctness condition is different (**coherence**). Namely, if the term can already be constructed, it obviously exists and is unique, but it may be necessary to prove that the term has the declared type.

The definitional theorem in that case has a simpler form:

```
for X,Y,Z being set holds
Z = X \+\ Y iff Z = (X \ Y) \/ (Y \ X);
```

Please note that although the definitional theorem is always generated by the system and may be referenced in proofs concerning that notion, it is often not necessary to refer to it. That is because the CHECKER uses *equals expansion* and every occurrence of such a term (like `X \+\ Y` in our example) may be automatically expanded to its full form (in that case `(X \ Y) \/ (Y \ X)`). The expansion takes place if the name of the article that contains such a definition is listed in the **definitions** environment directive (see Section 4.4). Please note that the current implementation of the Mizar system restricts the expansions to three levels only.

2.3.5 *Structures. Structures* in Mizar can be used to 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, e.g.:

```
definition
  struct multMagma
  (# carrier -> set,
     multF -> BinOp of the carrier #);
end;
```

where `multMagma` is the name of a structure with two selectors: an arbitrary set called its `carrier` and a binary operation on it, called `multF`. This structure can be used to define a group, but also upper and lower semilattices, so in fact any notion that is based on a set and a binary operation on it.

Please note that the above structure does not define a group yet (nor any other more concrete object), because there is no information on the properties of `multF`. The structure is just a basis for developing a theory. In practice, after introducing a required structure, a series of attributes is also defined to describe the properties of certain fields.

As mentioned before, the above `multMagma` structure can be used to define notions which are not only groups. Still, the operation in such structures inherit the name `multF`, because the current Mizar implementation does not provide a mechanism to introduce synonyms for selectors (or whole structures). Therefore, in cases when a different name is frequently used in standard mathematical practice, it may be better to introduce a different structure. For example, lattice operations are commonly called `meet` and `join`, so a lower semilattice may be better encoded as:

```
definition
  struct /\-SemiLattStr
  (# carrier -> set,
     L_meet -> BinOp of the carrier #);
end;
```

Mizar supports multiple inheritance of structures that makes a whole hierarchy of interrelated structures available in the Mizar library, with the `1-sorted` structure being the common ancestor of almost all other structures. For example, formalizing topological groups in Mizar can be done by independently defining and developing group theory and the theory of topological spaces, and then merging these two theories together based on a new structure, e.g.:

```
definition
  struct (1-sorted) TopStruct
  (# carrier -> set,
     topology -> Subset-Family of the carrier #);
end;

definition
  struct (multMagma, TopStruct) TopGrStr
  (# carrier -> set,
     multF -> BinOp of the carrier,
     topology -> Subset-Family of the carrier #);
end;
```

The advantage of this approach is that all notions and facts concerning groups and topological spaces are naturally applicable to topological groups.

Let us note that when introducing a new structure, the inherited selectors can be listed in any order, as far as relations between them are preserved. The list of names of ancestor structures is put in brackets before the name of the structure being defined.

An important extension of the Mizar structure system is the possibility to define structures parameterized by arbitrary sets, or other structures, e.g.:

```
definition
  let F be 1-sorted;
  struct (addLoopStr) VectSpStr 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;
```

For example, the above structure can be used as the basis of a vector space over the field of real or complex numbers. The system allows for defining structures with several parameters, all of them must be introduced as loci in the definition.

Concrete mathematical objects, e.g. the additive group of integers are introduced with so called *aggregates* - special term constructors defined automatically by the definition of a structure, e.g.: `multMagma(#INT,addint#)`, where `INT` is the set of integers, and `addint` represents the addition function. It is necessary that all terms used in the aggregate have the respective types declared in the structure's definition. In our example `INT` is obviously a set, and `addint` must be of type `BinOp of INT`.

Every structure defines implicitly a special attribute, **strict**. The corresponding adjective's meaning is that an object of a structure type contains nothing more, but the fields defined for that structure. For example, a term with structural type based on `TopGrStr` may be **strict** `TopGrStr`, but it is neither **strict** `multMagma`, nor **strict** `TopStruct`. Clearly, every term constructed using a structure's aggregate is **strict**.

Finally, the Mizar language has means to restrict a given term with a complex structure type to its well-defined subtype. This special term constructor, the *forgetful functor* also utilizes the structure's name, e.g. `the multMagma of G`, where `G` has a potentially wider type which inherits the `multMagma` structure. Again, such terms are **strict**, with respect to the given structure type.

2.3.6 *Synonyms and antonyms.* Mizar supports using synonyms of a relation, adjective, type or operation if a different symbol should be used instead of the original one. The new symbol must, of course, be appended to a vocabulary file.

Synonyms are introduced in the **notation** ... **end;** block that resembles the definitional block, e.g.:

```
definition
  let A be set, B be Subset of A;
  pred pred_symbol_1 A,B means not contradiction;
end;

notation
  let A be set, B be Subset of A;
  synonym pred_symbol_2 A,B for pred_symbol_1 A,B;
end;
```

The types of loci in a synonym declaration may be more specific than those in the original definition. However, it is not possible to substitute loci with constant terms, so a set `A` cannot be substituted with the set of natural numbers `NAT`:

```
notation
  let B be Subset of NAT;
  synonym pred_symbol_2 B for pred_symbol_1 B,NAT;
::>                                          *300,142
::> 142: Unknown locus
::> 300: Identifier expected
end;
```

Synonyms can also be used to change the format, e.g. from prefix to infix like below:

```
notation
  let A be set, B be Subset of A;
  synonym A pred_symbol_2 B for pred_symbol_1 A,B;
end;
```

Moreover, a synonym can change (increase or decrease) the number of visible arguments, e.g.:

```
notation
  let A be set, B be Subset of A;
  synonym pred_symbol_1 B for pred_symbol_1 A,B;
end;
```

Please note that, like in definitions, here also one must obey the rules that disallow repeated or unused loci.

In the case of modes, synonyms can only be used for non-expandable modes. Therefore the following notation would be incorrect:

```
notation
  let X be set;
  synonym mode_symbol of X for Subset of X;
::>                                  *134
end;
::> 134: Cannot redefine expandable mode
```

because `Subset of X` is defined as an expandable mode (`Element of bool X`).

An important feature of Mizar synonyms is that they do not inherit the type of a redefined original constructor (see Section 2.4 for more information on redefinitions). Let us take a look at an example where `Morphism of a,b` denotes an arbitrary morphism between objects of the same category. In the following example, the type is defined with the mother type `set`:

```
definition
  let C be non void non empty CatStr, a,b be Object of C;
  mode Morphism of a,b -> set means
  ...
end;
```

and then the type is redefined as `Morphism of C` (any morphism in the underlying category):

```
definition
  let C be non void non empty CatStr, a,b be Object of C;
  redefine mode Morphism of a,b -> Morphism of C;
  ...
end;
```

If we consider a synonym:

```
notation
  let C be non void non empty CatStr, a,b be Object of C;
  synonym morphism of C,a,b for Morphism of a,b;
end;
```

then the synonym does not possess the type `Morphism of C`, as seen in the following proof extract:

```
set C = the non void non empty CatStr;
set a = the Object of C;
set b = the Object of C;
set m = the morphism of C,a,b;
m is Morphism of a,b;
m is Morphism of C;
::>              *4
::> 4: This inference is not accepted
```

Analogous rules apply to introducing synonyms for functors. In the case of predicates and attributes, one may also introduce antonyms. The rules concerning introducing antonyms are the same, so let us only present here one example of their use:

```
notation
  let i be Integer;
  antonym i is odd for i is even;
end;
```

Then instead of using the artificially looking phrases like `non even Integer` one may use a much more natural notation.

The paper [7] provides useful hints to make definitions effective.

2.4 Redefinitions

Redefinitions are used to change the definiens or type for some constructor if such a change is provable with possibly more specific arguments (please note that structure definitions cannot be redefined). Depending on the kind of 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. A detailed description can be found in Appendix A.

Considering the most general notion of equality:

```
definition
  let x, y be set;
  pred x = y;
end;
```

the predicate may be better described in the context of specific arguments. For example, if the arguments of equality are known to be relations, then the fact that they are equal may be expressed using the notion of pairs:

```
definition
  let P, R be Relation;
  redefine pred P = R means
  for a,b being set holds [a,b] in P iff [a,b] in R;
end;
```

Similarly, equality of functions is best expressed if we can compare the results of their application on the common domain:

```
definition
  let f, g be Function;
  redefine pred f = g means
  dom f = dom g & for x being set st x in dom f holds f.x = g.x;
end;
```

Redefinitions may also be used to introduce properties (see Section 2.5) when the property does not hold for the original (more general) predicate. For example, with the following generic operation:

```
definition
  let M be multMagma;
  let x, y be Element of M;
  func x*y -> Element of M equals
  (the multF of M).(x,y);
end;
```

the result need not be commutative. But if the underlying structure is appropriate (commutative in this redefinition), the property can be recorded:

```
definition
  let M be commutative non empty multMagma;
  let x, y be Element of M;
  redefine func x*y;
  commutativity;
end;
```

It must be noted here that Mizar uses a *flat* concept of redefinitions, i.e. a redefinition always redefines the original constructor and never a previously introduced redefinition.

Another important restriction is that only a proper pattern can be used in a redefinition. In other words, it is not possible to redefine a constructor with a loci substituted with a constant. For example, with the following definition:

```
definition
  let X, Y be set;
  func X \/ Y -> set means
  x in it iff x in X or x in Y;
end;
```

it is not allowed to make a redefinition with the constant term `NAT` used in place of a locus:

```
definition
  let X be set;
  redefine func X \/ NAT -> non empty set;
::>                *113
  coherence;
end;
::> 113: Unknown functor
```

Redefinitions that change the type of a constructor may only be used when the new type widens to the original one. Let us consider the definition below:

```
definition
  let f be Function;
  assume f is one-to-one;
  func f" -> Function equals
  f~;
end;
```

Then an example of a valid redefinition may be one where the argument is a permutation, and so is the new result type:

```
definition
  let X be set;
  let f be Permutation of X;
  redefine func f" -> Permutation of X;
end;
```

Restricting the original type is, therefore, incorrect, so the following redefinition would yield an error:

```
definition
  let X be set;
  let f be Permutation of X;
  redefine func f" -> set;
::>                     *117
  coherence;
end;
::> 117: Invalid specification
```

Please also note that it is required that the order of loci used in a redefinition match the original definition. For example, with the definition:

```
definition
```

```
  let f be Function, x be set;
  func f.x -> set means
  [x,it] in f if x in dom f otherwise it = {};
end;
```

one might want to redefine it in the following more specific context, however, an error would be reported:

```
definition
  let C be non empty set, D be set;
  let c be Element of C;
  let f be Function of C,D;
  redefine func f.c -> Element of D;
::>              *109
end;
::> 109: Invalid order of arguments of redefined constructor
```

In this case, the solution would be to change the order of `f` and `c`, as follows:

```
definition
  let C be non empty set, D be set;
  let f be Function of C,D;
  let c be Element of C;
  redefine func f.c -> Element of D;
end;
```

The above example shows that extra loci may be added compared to the original. The current Mizar implementation does not allow to reduce the number of loci, even if the types of the available (fewer) arguments might widen to the types in the definition.

2.5 Properties

As stated in Section 2.3, some of the defined notions can be accompanied in their definitional block by statements and proofs of specific *properties*. These properties are stored together with the definitions, so the Mizar CHECKER can automatically use a corresponding formula as an extra heuristic whenever the notion is used. Some of the properties may be introduced in redefinitions (see Section 2.4), where loci types are more specific.

2.5.1 *Projectivity.* This is the property of a function which states that its double application does not change the result, i.e f(f(x)) = f(x). A typical example of a functor definition which possesses this property is:

```
definition
  let x be real number;
  func sgn x -> integer number equals
  1 if 0 < x, -1 if x < 0 otherwise 0;
  ...
  projectivity;
end;
```

The **projectivity** property is only applicable to functors with one visible argument, and the result type must widen to the type of the argument. If the property is stated, a corresponding correctness condition must be proved within the definition (see Appendix A for more details). In the current Mizar implementation, this property cannot be used in redefinitions.

2.5.2 *Involutiveness.* This is the property of a function which says that the result of double application is equal to the original argument, i.e. f(f(x)) = x. A typical example would be an operation of taking the inverse element, e.g. in a group:

```
definition
  let G be Group, h be Element of G;
  func h" -> Element of G means
  h * it = 1_G & it * h = 1_G;
  ...
  involutiveness;
end;
```

Here it is also meaningful only if there is one visible argument. The result type is required to be equal to the type of the argument. The correctness condition to be proved for definitions with **involutiveness** can be found in Appendix A. In the current Mizar implementation, this property cannot be used in redefinitions.

2.5.3 *Idempotence.* An idempotent binary operation applied to two equal values gives that value as the result. For example the set-theoretical union has this property (`x \/ x = x`):

```
definition
  let X,Y be set;
  func X \/ Y -> set means
  x in it iff x in X or x in Y;
  ...
  idempotence;
end;
```

The **idempotence** property is applicable to functors with two visible arguments with the same type. The correctness condition to be proved for definitions with **idempotence** can be found in Appendix A. In the current Mizar implementation, this property cannot be used in redefinitions.

2.5.4 *Commutativity.* This is a property of binary operations that allows to simply swap the arguments. Again, the set theoretical union obviously has this property (`x \/ y = y \/ x`):

```
definition
  let X,Y be set;
  func X \/ Y -> set means
  x in it iff x in X or x in Y;
  ...
  commutativity;
```

```
end;
```

The **commutativity** property is applicable to functors with two visible arguments with the same type. See Appendix A for more information on the required correctness condition. Please note that **commutativity** can be used in redefinitions.

Below we present five kinds of supported properties that are applicable to binary predicates (and can be introduced in both definitions and redefinitions).

2.5.5 *Reflexivity.* A relation is said to be reflexive if any argument is in that relation with itself. Let us take as an example the divisibility relation:

```
definition
  let i1,i2 be Integer;
  pred i1 divides i2 means
  ex i3 being Integer st i2 = i1 * i3;
  reflexivity;
end;
```

See Appendix A for the description of the formula that needs to be proved as the correctness condition.

2.5.6 *Irreflexivity.* Conversely, a relation is irreflexive if any argument can never be in that relation with itself, as is the case with e.g. the proper inclusion:

```
definition
  let X,Y be set;
  pred X c< Y means
  X c= Y & X <> Y;
  irreflexivity;
end;
```

Appendix A presents the formula that needs to be proved as the correctness condition.

2.5.7 *Symmetry.* With a symmetric relation the arguments can be swapped. An example of a clearly symmetric predicate is presented below:

```
definition
  let X,Y be set;
  pred X misses Y means
  X /\ Y = {};
  symmetry;
end;
```

See Appendix A for the description of the formula that needs to be proved as the correctness condition.

2.5.8 *Asymmetry.* Asymmetric predicates are ones that always change their logical value whenever the arguments are swapped. An example we used to demonstrate **irreflexivity** can also be used here, as the proper inclusion is obviously asymmetric, too:

```
definition
  let X,Y be set;
  pred X c< Y means
  X c= Y & X <> Y;
  asymmetry;
end;
```

The corresponding correctness condition to be proved is presented in Appendix A.

2.5.9 *Connectedness.* A binary relation is said to be connected if for any arguments `a` and `b`, either `(a,b)` or `(b,a)` is a member of the relation. A typical example is the inclusion relation defined on ordinal numbers:

```
definition
  let A,B be Ordinal;
  redefine pred A c= B means
  for C st C in A holds C in B;
  connectedness;
end;
```

See Appendix A for more information on the corresponding correctness condition.

2.6 Registrations

In the Mizar language, the common name *registration* refers 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 (see [17] for a detailed description of this mechanism).

The first kind are the *existential* registrations which are used to secure the non-emptiness of Mizar types. For example, the following registration provides a proof (stated as the **existence** correctness condition) that there exists at lease one set which is both finite and non-empty:

```
registration
  cluster finite non empty set;
  existence
  ...
end;
```

Another kind of registrations is called a *conditional* registration. Let us demonstrate its syntax by the following example which states that every set which is empty must also be finite:

```
registration
  cluster empty -> finite set;
  coherence
  ...
end;
```

Let us note here that the Mizar syntax allows the list of adjectives before the `->` arrow to be empty. In that case a registration is used to record adjectives which can always be associated with the given type. For example:

```
registration
  let X be finite set;
  cluster -> finite Subset of X;
  coherence
  ...
end;
```

The dependencies of adjectives recorded as conditional registrations are then used automatically by the Mizar verifier. However, it is important to distinguish their processing in the ANALYZER and the CHECKER modules. The calculus done within the CHECKER (see Section 3.5) makes it possible to infer some consequences of registrations e.g. a contraposition of a conditional registration. This is not possible in the ANALYZER, so the type checking done in this module may in some cases require an explicitly stated registration:

```
cluster infinite -> non empty set;
```

although this consequence of adjectives is obvious for the CHECKER (`infinite` is defined as an antonym for `finite`).

The users must be aware of the fact that existential registrations are extended with the available conditional consequences. For example, with the two following conditional registration available:

```
cluster empty -> Relation-like set;
```

```
cluster empty -> Function-like set;
```

the subsequent existential registration:

```
cluster empty set;
```

registers in fact the following type with the (*rounded-up*) cluster of adjectives:

```
cluster empty Relation-like Function-like set;
```

Knowing that feature is extremely important, because a successful import of such a registration from a database is only possible if the constructors of all its adjectives are available in the article's environment (the CONSTR utility may be used to detect all needed constructors for a particular registrations with respect to a given environment, see Section 4.5).

Finally, the third kind of registrations are *term adjectives registrations*, sometimes also called *functorial registrations* in the Mizar jargon. As the name suggests, they are used to register a certain property possessed by a specified term. In contrast to redefinitions, not only patterns can be used, but also more complex terms. The term may be constructed using a functor, selector, aggregate or forgetful functor. For example:

```
registration
  let X be non empty set;
  let Y be set;
  cluster X \/ Y -> non empty;
  coherence
  ...
end;
```

An extended syntax of term adjectives registrations (with an extra type) can be used to register adjectives which are applicable only for terms with a redefined type. Let us consider the following registration:

```
registration
  let T be TopSpace;
  let X, Y be open Subset of T;
  cluster X \/ Y -> open Subset of T;
  coherence
  ...
end;
```

In this case, the adjective `open` is only correct if the type of the term (`X \/ Y`) is properly identified as the redefined variant (`Subset of T`).

Please bear in mind that it is a common misconception (resulting from similar syntax) that registrations with the extra type change the type of the registered term the way that redefinitions do. On the contrary, to accept such a registration, the system must be able to infer that the given term has the provided type (as a result of a respective redefinition).

Let us finally state here as a general rule that registrations are usually preferred to redefinitions. The first reason is that registrations do not introduce new constructors. Also the adjective clusters imported via registrations are simply cumulated, while the *flat* concept of redefinitions gives access only to the last redefinition available. More information on the internal processing of adjectives in Mizar can be found in [11].

2.7 Terms identification

In mathematical practice, a given object or an operation is often treated in many different ways depending on contexts in which they occur. A natural number can be considered as a von Neumann number, or as a finite ordinal. The least common multiply can be considered as an operation on numbers, or as the supremum of elements of some lattice, and so on. In such cases it is often worthwhile to have automatic 'translation' theorems. In Mizar this is done with *terms identification* using the **identify** keyword (technically implemented in the registration block), e.g.:

```
registration
  let p, q be Element of Nat_Lattice;
  identify p "\/" q with p lcm q;
  compatibility;
end;
```

The aim of the identification is matching the term at the left side of the **with** keyword with the term stated at the right side, whenever they occur together. The current implementation allows matching in one direction only, i.e. when the verifier processes a sentence containing the left side term, it generates its local copy with the left side term symbol substituted by the right side one and makes both terms equal to each other. Such an equality allows to justify facts about the left side terms via lemmas written about the right side ones, but not vice versa. In this

sense identification is not symmetric, which is showed with the following example, where `Nat_Lattice` is the lattice of naturals with `lcm` and `hcf` as operations. Then, the following lemma

```
L1: for x, y, z being natural number holds
    x lcm y lcm z = x lcm (y lcm z);
```

can be used to justify the sentence

```
L2: for x, y, z being Element of Nat_Lattice holds
    x "\/" y "\/" z = x "\/" (y "\/" z) by L1;
```

but justifying `L1` with `L2` directly does not work.

Terms identification is available immediately at the place where it is introduced till the end of the article. If one wants to use the identification introduced in an external article, it should be imported in the environment. The current implementation of identification is internally similar to registrations, so identification does not have a library directive on its own, so identifications are imported with **registrations**.

Below are typical errors that may be reported while working with term identifications:

```
registration
  let p, q be Element of Nat_Lattice;
  identify p "\/" q with 1_NN;
::>                          *189,189
end;

::> 189: Left and right pattern must have
         the same number of arguments
```

In this case the error description offered by the checker is self-explanatory.

```
registration
  let p, q be Element of Nat_Lattice;
  let m, n be Nat;
  identify p "\/" q with m lcm n when p = m, q = n;
::>                                      *139   *139
end;

::> 139: Invalid type of an argument.
```

This error means that the types of variables `p` and `q` do not round up to the types of `m` and `n`, respectively. The solution to the problem is the registration:

```
registration
  cluster -> natural Element of Nat_Lattice;
  coherence;
end;
```

2.8 Summary of definitions, redefinitions and registrations

The table below summarizes the usage of required correctness conditions and properties applicable to a particular kind of definition.

Definitions		
	Correctness conditions	Properties
Predicate	—	**reflexivity** **irreflexivity** **symmetry** **asymmetry** **connectedness**
Attribute	—	—
Mode	**existence**	—
Functor	**existence** **uniqueness**	**commutativity** **idempotence** **involutiveness** **projectivity**

In case of redefinitions the required correctness conditions and properties depend on whether the result type or definiens is changed:

Redefinitions changing the result type		
	Correctness conditions	Properties
Mode	**coherence**	—
Functor	**coherence**	**commutativity**

Redefinitions changing the definiens		
	Correctness conditions	Properties
Predicate	**compatibility**	**reflexivity** **irreflexivity** **symmetry** **asymmetry** **connectedness**
Attribute	**compatibility**	—
Mode	**compatibility**	—
Functor	**compatibility**	**commutativity**

Below is a list of correctness conditions required to justify a particular kind of registration.

Registrations	
	Correctness conditions
Existential	**existence**
Conditional	**coherence**
Functorial	**coherence**
Term identification	**compatibility**

It should also be observed that partial definitions require an additional **consistency** condition (see Appendix A). Moreover, please note that a universal correctness condition **correctness** can be typed as a replacement for all correctness conditions remaining to be proved in a given definition or registration.

3. SYSTEM

The Mizar verifier consists of several modules responsible for checking various aspects of correctness of Mizar articles. They are: SCANNER, PARSER, ANALYZER, REASONER, CHECKER, SCHEMATIZER.

3.1 Scanner – tokenizer

The SCANNER reads the source file and slices it into tokens. As the first step, all comments (parts of lines that start with the double colon `::`) are pruned. Next the module analyzes segments of the text delimited by whitespaces and cuts them into tokens – the longest possible strings that can be classified into one of the categories:

reserved words. – the full list is provided in Section 2.

symbols. – introduced in users' vocabulary files to denote defined notions; a symbol may contain any characters of the 7-bit ASCII code with codes above 32.

numerals. – sequences of digits starting with a non-zero digit (hence `0` is not treated as a numeral); the numeric value of a numeral cannot exceed 32767 (maximal signed 16-bit integer).

identifiers. – strings of letters or digits that are neither reserved words, symbols, nor numerals; identifiers are used to name variables, schemes and labels; according to that rule, the text `01` is a valid identifier.

filenames. – uppercase strings of up to eight characters (alphanumeric and underscores); used in environment directives to import items from selected articles, and in library references within the main part of an article.

Let us finally recall here that the case of letters is significant for Mizar.

3.2 Parser

The main role of the Mizar PARSER is checking the syntactic correctness with respect to the grammar (see Appendix B) of the stream of tokens produced by the SCANNER and producing the abstract representation of the article in the form of stacked blocks and items, to be used by the ANALYZER.

It must be noted that the representation is generated taking into account some features of the language that are not regulated by the rules of grammar: the priority and arity of imported operations, and a dedicated algorithm for long term analysis (see [19] for a detailed description). To reduce the number of parentheses, additional rules specify that the associative notation can be used for conjunctive and disjunctive formulas, but not for implications and equivalences. Moreover, the binding force of logical connectives is stronger than that of quantifiers.

3.3 Analyzer

The main role of the ANALYZER is identification (disambiguation) of used constructors and notations on the basis of the type information imported from the environment and available registrations.

If more than one way of identification is possible (as a result of symbol overloading), the one that is last in the **notations** directive is effective. Let us assume, for example, that we have two addition operations denoted with the same + symbol, one for complex numbers and the other for extended real numbers (with $\pm\infty$).

Then the expression `1+2` could be understood in two ways, provided the system is able to infer that the numbers `1` and `2` have both types. The Analyzer would then choose the operation which comes last in the list of available notations.

There is, however, a mechanism available in Mizar that can be used to force using a selected identification. With the above example, the phrase

```
1 qua complex number + 2
```

forces the Analyzer to treat `1` as a complex number, and as a consequence the variant for extended reals cannot be used.

3.4 Reasoner

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.

Thanks to that simplification, a skeleton of a proof is considered valid as far as the semantic correlate it generates is the same as that of the statement being proved.

3.5 Checker

The most complex module of the Mizar system is its Checker that works as a classical disprover. Most importantly, in that module an inference of the form

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

is transformed to

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

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

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

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

$$\frac{[\neg]\alpha^{1,1} \wedge \ldots \wedge [\neg]\alpha^{1,k_1}}{\bot}$$
$$\ldots$$

$$\frac{[\neg]\alpha^{n,1} \wedge \ldots \wedge [\neg]\alpha^{n,k_n}}{\bot}$$

In the refutation process, the CHECKER uses numerous heuristics that combine into its notion of *obvious inference*. The processing concerns the type information associated with all terms, the properties of used constructors, the equality calculus, equals expansion and term identifications (**identify** registrations). The CHECKER also uses special built-in automation procedures for processing selected, very frequently used objects like e.g. complex numbers (direct computation) or boolean operations on sets. These internal routines turned on using the **requirements** directive are described in detail in [13].

3.5.1 *Schematizer.* Mizar supports the so called *schemes* to enable feasible encoding of statements frequently used in standard mathematics that go beyond first-order logic. In Mizar one may use free second order variables for forming schemes of theorems (infinite families of theorems). The syntax is best seen with a typical example, e.g. the induction scheme:

```
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];
```

If the second order variable represents a predicate, it is denoted by an identifier followed by a list of argument types in square brackets (e.g. `P[Nat]`). In case of a functor, simple brackets are used and the result type is stated (e.g. `F(Nat) -> Nat`). Functors with an empty list of arguments are called *scheme constants*.

Locally, (within the article that introduces a scheme) a scheme is referenced by its label (`NatInd` in this example). A scheme stored in the Mizar database can be referenced with a library reference. To use a scheme one needs to follow these three steps:

(1) define a private functor or predicate to be pattern-matched with the scheme's variables, depending on the scheme
(2) prove the scheme's premises
(3) make a reference to the scheme using the **from** keyword in a justification of the statement (an external scheme must be imported with the `schemes` environment directive and the order of premises must match exactly the scheme's declaration).

An example of scheme usage that demonstrates the three steps marked with respective comments is presented below:

```
2 divides n * (n+1)
proof
  :: step 1 - private predicate
  defpred P[Nat] means 2 divides $1 * ($1+1);
  :: step 2 - premises
a1: P[0];
a2: for k being Nat st P[k] holds P[k+1];
```

```
  :: step 3 - referring to the scheme
  for k being Nat holds P[k] from NAT_1:sch 2(a1,a2);
  hence 2 divides n * (n+1);
end;
```

Below we point out typical errors that may be encountered while using schemes. Let us assume that we have a scheme:

```
scheme :: NAT_1:sch 12
  { D() -> non empty set, A() -> Element of D(),
    G(set,set) -> Element of D() }:
  ex f being Function of NAT,D() st f.0 = A() &
     for n being Nat holds f.(n+1) = G(n,f.n);
```

and want to use it to prove the existence of a recursive function, e.g.

```
definition
  let s be Real_Sequence;
  func Partial_Sums(s) -> Real_Sequence means
:: SERIES_1:def 1
  it.0 = s.0 & for n being Nat holds it.(n+1) = it.n + s.(n+1);
end;
```

The correct usage may look like:

```
deffunc U(Nat,Real) = $2 + s.($1+1);
consider f being Function of NAT,REAL such that
f.0 = s.0 & for n being Nat holds f.(n+1) = U(n,f.n)
from NAT_1:sch 12;
```

where the scheme constants `D()` and `A()` are substituted by `REAL` and `s.0`, and the scheme variable `G(set,set)` by the local functor `U`, respectively. This works, because the types of the substituting terms fit the scheme declaration, i.e. `REAL` has the type `non empty set`, `s.0` is `Element of REAL`, and the result of `U` is `Element of REAL` as well.

If we tried to use some irrelevant `g` instead of `f`, an error would be reported:

```
consider f being Function of NAT,REAL such that
g.0 = s.0 & for n being Nat holds f.(n+1) = U(n,f.n) from NAT_1:sch 12;
::>                                                      *20
::> 20: The structure of the sentences disagrees with the scheme
```

Similarly, if instead of `s.0` some other term was used, say `s`, that does not have the expected type (`Element of REAL` in this example), an error would occur:

```
consider f being Function of NAT,REAL such that
f.0 = s & for n being Nat holds f.(n+1) = U(n,f.n) from NAT_1:sch 12;
::>                                                    *26
::> 26: Substituted constant does not expand properly
```

Again, using e.g. `s` (without a proper type) instead of `$2 + s.($1+1)` in the **deffunc** definition would also be invalid:

```
deffunc U(Nat,Real) = s;
consider f being Function of NAT,REAL such that
f.0 = s.0 &
for n being Nat holds f.(n+1) = U(n,f.n) from NAT_1:sch 12;
::>                                          *30
::> 30: Invalid type of the instantiated functor
```

As stated before, to use a scheme we first define a private functor or predicate to be pattern-matched with the scheme's variables. This step is not necessary, however, when a scheme predicate or functor is substituted by a matching formula or term with exactly the same number and order of arguments. Let us analyze the following example where we want to construct a function $f : \mathbb{N} \to \mathbb{N}$ such that $f(n) = n!$. One can use the following scheme:

```
scheme :: FUNCT_1:sch 4
  Lambda { D() -> non empty set, F(set) -> set } :
  ex f being Function st dom f = D() &
    for d being Element of D() holds f.d = F(d);
```

substituting `D()` with the set of natural numbers `NAT` and `F(set)` with a local functor:

```
deffunc F(Element of NAT) = $1!;
```

to get the function `f` as below

```
ex f being Function st dom f = NAT &
for d being Element of NAT holds f.d = F(d) from Lambda;
```

But, because both `F(set)` and `!` are unary functors, the same result can be obtained directly as:

```
ex f being Function st dom f = NAT &
for d being Element of NAT holds f.d = d! from Lambda;
```

However, if we wanted to introduce $f : \mathbb{N} \to \mathbb{N}$ such that $f(n) = 2 * n$, this method would not work:

```
ex f being Function st dom f = NAT &
for d being Element of NAT holds f.d = 2*d from Lambda;
::>                                        *28
::> 28: Invalid list of arguments of a functor
```

The error is generated because the multiplication (`*`) is a binary operation while `F(set)` is expected to have just one argument. In this case defining a private functor is required to use the scheme.

```
deffunc G(Element of NAT) = 2*$1;
ex f being Function st dom f = NAT &
for d being Element of NAT holds f.d = G(d) from Lambda;
```

Private definitions are also necessary when the order of arguments does not match exactly. For example, to create $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that $f(m, n) = m^n$ one could use the following scheme:

```
scheme :: BINOP_1:sch 4
  Lambda2D { X,Y,Z() -> non empty set,
             F(Element of X(),Element of Y()) -> Element of Z() } :
  ex f being Function of [:X(),Y():],Z() st
  for x being Element of X() for y being Element of Y()
    holds f.(x,y) = F(x,y);
```

as follows

```
ex f being Function of [:NAT,NAT:],NAT st
for x,y being Element of NAT holds f.(x,y) = x |^ y from Lambda2D;
```

But if we wanted to have $f(m, n) = n^m$, an error would be reported:

```
ex f being Function of [:NAT,NAT:],NAT st
for x,y being Element of NAT holds f.(x,y) = y |^ x from Lambda2D;
::>                                                        *28
::> 28: Invalid list of arguments of a functor
```

because the order of `x` and `y` is reversed. To use the scheme in this situation, a private functor like

```
deffunc F(Element of NAT,Element of NAT) = $2 |^ $1;
```

must be applied as below:

```
ex f being Function of [:NAT,NAT:],NAT st
for x,y being Element of NAT holds f.(x,y) = F(x,y) from Lambda2D;
```

Similar restrictions concern the use of scheme predicates.

4. SOFTWARE

4.1 Installation

The Mizar system and the Mizar Mathematical Library are publicly available for download and may be used by anyone free of charge for non commercial purposes. The contents of the Mizar distribution is copyrighted by the Association of Mizar Users which maintains the Mizar Mathematical Library and coordinates the development of Mizar software. The system is available in a ready to use precompiled form for many operating systems. Latest Mizar releases (version 7.11.07, MML 4.156.1112) can be downloaded from the Mizar site via anonymous FTP or HTTP for the following OS's:

—Linux (i386),
—Solaris (i386),
—FreeBSD (i386),
—Darwin/Mac OS X (i386),
—Darwin/Mac OS X (PPC),
—Linux (PPC),
—Linux (ARM),
—Win32.

The distribution is self-contained, so the installation is rather straightforward. The package contains the Mizar processor, all the articles constituting the Mizar Mathematical Library and their abstracts, the database based on these articles, a set of utility programs, and a GNU Emacs Lisp mode for convenient work with the system. The installation requires about 220 MB of free disk space.

Below we show how to install Mizar in a Unix-like environment, and also on Microsoft Windows.

4.1.1 *Unix-like OS's.* The distribution can be downloaded as a `tar` archive, e.g. `mizar-7.11.07_4.156.1112-i386-linux.tar`. The name of the archive indicates the version of the Mizar system it contains (here `7.11.07`), the version of MML (here `4.156.1112`) and the OS the software has been compiled for (here an i386-based Linux).

Within the archive there is a script `install.sh` which is used to unpack the Mizar system to specified directories. In most cases it is enough to call it without any parameters:

```
./install.sh
```

and then answer three questions about the place where files should be copied. Please note that the installation of a new version always replaces an old one completely if the same directories are specified during both installations to avoid any version conflicts.

There are two options that may be used with this script:

```
./install.sh --default
```

runs the script in a non-interactive mode (see below for default directories),

```
./install.sh --nodialog
```

always runs the script in a plain mode (not using the `dialog` utility to a semi-graphic user interface for the installation) getting input directly from STDIN (useful for using within shell scripts).

When running the installation script in the interactive mode, first you must provide a name of a directory where you want to install Mizar executables (default is `/usr/local/bin`). Make sure this directory is in your `PATH` environment variable.

Then you will be prompted to choose a directory for Mizar shared files (default is `/usr/local/share/mizar`).

Please note that after the installation you must set a new environment variable, `MIZFILES` to point to this directory. For example, when using the Bash shell, you may put the following line in the Bash config file, `.bashrc` to have the setting permanent:

```
export MIZFILES=/home/john/my-mizar-files
```

Otherwise, the system will not be able to find the database correctly.

Finally you will be asked by the installation script about a directory where Mizar documentation files should be placed (default is `/usr/local/doc/mizar`).

After that, with the `MIZFILES` set correctly, the system is ready to use. The distribution contains a GNU Emacs Lisp initialization file which you may use for convenient work with the system. To use it, simply append the file `.emacs` from the selected Mizar `doc` directory to your GNU Emacs initialization file, usually stored as the `.emacs` file in a user's home directory.

4.1.2 *Microsoft Windows.* The distribution can be downloaded as a self-extracting `zip` archive e.g. `mizar-7.11.07_4.156.1112-i386-win32.exe`. The name of the archive indicates the version of the Mizar system it contains (here `7.11.07`), the version of MML (here `4.156.1112`). The software has been precompiled so that it can be run on virtually any version of Microsoft Windows (9x/2000/NT/XP/Vista/7).

Executing the self-extracting archive in a temporary directory produces a number of compressed archives and an install script `INSTALL.BAT` which is used to unpack the Mizar system to a specified directory. In most cases it is enough to call the script at the command prompt set in that directory with one directory parameter, e.g.:

```
INSTALL c:\mizar
```

The path `c:\mizar` can be replaced with the name of a different directory and a different hard drive letter where the system should be installed. If the path is not given, the default `c:\mizar` will be used. Please note that the installation of a new version always replaces an old one completely if the same directories are specified during both installations to avoid any version conflicts.

Please also note that after the installation you must set a new environment variable, `MIZFILES` to point to the given directory. Otherwise, the system will not be able to find the database correctly.

To be able to run Mizar executables without specifying the full path, you may also want to append the system PATH variable with the name of that directory. In newer Windows versions these tasks can be done by editing an appropriate entry in the `Environment Variables` tab in the `Control Panel -> System -> Advanced` menu.

On older Windows systems which utilize the `AUTOEXEC.BAT` configuration file, one could add the following lines to that file to have the same effect (replacing `c:\mizar` with a different directory if necessary):

```
set PATH=c:\mizar;%PATH%
set MIZFILES=c:\mizar
```

After that, with the `MIZFILES` set correctly, the system is ready to use. The distribution contains a GNU Emacs Lisp initialization file which you may use for convenient work with the system. To use it, simply append the file `.emacs` from the Mizar `doc` subdirectory to your GNU Emacs initialization file, usually stored as the `.emacs` file in a user's home directory (pointed to by the environment variable

HOME or the HKCU\SOFTWARE\GNU\Emacs\HOME registry entry) or at the root of the main file system (c:\.emacs).

4.2 Preparing a Mizar article

The mechanics of preparing a Mizar article is as follows:

—The source text is prepared using any ASCII editor and typically includes from 1500 to 5000 lines.

—The text is run through the Accommodator. The directives from the environment declaration guide the production of the environment specific for the article. The environment is produced from the available database.

—Now the Verifier is ready to start checking. The output contains remarks on unaccepted fragments of the source text.

These three steps are repeated in a loop until no errors are flagged and the author is satisfied with the resulting text. Usually, Accommodator and Verifier are called within mizf (or mizf.bat) user script. Alternatively, Josef Urban's Emacs-Lisp Mizar mode (now included in all Mizar distributions) provides a fully functional interface to the Mizar system.

When finished, an article is submitted to the Library Committee of Association of Mizar Users for inclusion into the Mizar Mathematical Library. The contributed article is subject to a review and if needed the authors must revise their file. The contents of an accepted article is extracted by the Exporter and Transfer utilities and incorporated into the public database distributed to all Mizar users. Alternatively, the user can prepare a local database files by invoking the command

```
miz2prel article
```

and then

```
miz2abs article
```

to obtain also an abstract file with the extension .abs, i.e. the version of the .miz file without proofs. Data extracted by Exporter is stored in prel subdirectory of the current directory.

4.3 Vocabularies

Vocabularies define additional lexicons for Mizar articles. Like a Mizar article, a vocabulary file is a plain ASCII file. Although it is in fact independent of any article, its name should be the same as the name of an article the vocabulary is attached to. The necessary extension is .voc and it should be placed in the subdirectory dict of the directory with the .miz file (if it not placed in the text subdirectory as usually suggested). Each line of a vocabulary file introduces a symbol; it begins with one character qualifier which determines the kind of the symbol, followed immediately by the representation of the defined symbol which is a string of arbitrary characters excluding control characters, space, and double colon.

Qualifiers of the symbol have the following meaning:

—R – predicate,

—O – functor,

—M – mode,

—G – structure,

—U – selector,
—V – attribute,
—K – left functor bracket,
—L – right functor bracket.

For a functor symbol an additional information may be given in a vocabulary file, namely, the priority of the functor. The priority is an integer between 0 and 255, written on the same line after a functor's identifier and a space. If not specified, the priority is by default equal to 64. For example, to facilitate the conventional precedence of multiplication over addition, in the current MML the + symbol is introduced with the lower priority

```
0+ 32
```

while * uses the default value.

4.4 Accommodator and environment declaration

The environment declaration of a Mizar article consists of the following directives:

vocabularies. Imports all symbols from the listed vocabularies to be used by Scanner and Parser.

constructors. Imports all constructors defined in the listed articles and then 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 its type and in types of its arguments.

notations. Imports formats and patterns that are used by the already imported constructors provided all the constructors needed to understand the notation have already been imported.

registrations. Imports the definitions of registrations that state relationships among adjectives, modes and functors.

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

schemes. Similar to the above, gives access to schemes.

definitions. Requests definientia that can be used in proving by definitional expansion without mentioning the definition's name. It allows also for automatic expansion of functors defined by **equals**.

requirements. Gives access to the built-in features associated with certain special articles.

The Accommodator (Accom) gathers information from the Mizar database according to the above directives, preparing a bunch of auxiliary files needed for the Verifier. Alternatively, the Makeenv utility which resembles the Unix `make` can be used – it calls Accom only if the environment declaration was modified and intermediate files should be rebuilt.

4.5 Auxiliary utilities

Information about used symbols (**vocabularies**) is collected in a single text file in the Mizar distribution, `mml.vct`. To facilitate the use of this file some simple querying tools are attached. All these can be run from Emacs Menu: Mizar – Voc. Constr. Utilities.

FINDVOC. The most common calling sequence of this tool which finds the vocabulary containing queried symbol is

```
findvoc -w <symbol>
```

where the switch `-w` means matching whole words.

For every symbol, its priority (only in case of functor symbol) and its qualifier is given. No priority means a default one.

Because of the policy of the Library Committee to keep the symbol in the vocabulary with the article identifier of its first use (not to have disjoint name spaces for vocabularies and articles), this utility can give a rough view which environment directives should be extended.

LISTVOC. By calling this program (listing vocabularies) with the MML identifier `example` one obtains the list of all symbols introduced by the vocabulary `example`. As a rule, it is the first (taking into account the ordering given by `mml.lar`) article using this symbol, this may however be changed as a result of a revision.

CHECKVOC. This simple utility (checking vocabulary) should be invoked before the submission of the article for inclusion in the MML. It checks if the user's vocabulary does not contain any prohibited characters (see Section 4.3), warns if one-letter symbols are present or if there is a symbol introduced by someone else (i.e. already available in `mml.vct`), possibly even in different context.

CONSTR. Getting environment description right to properly identify library objects is a major difficulty, especially for beginners. Once the text is correct with respect to the ANALYZER, the produced XML format gives unambiguous information about origins of any object. Sometimes error `*190` (Inaccessible theorem) is marked.

```
constr ArticleName:n
```

gives the list of all constructor directives needed to recognize n-th theorem from the article `ArticleName`. If one wants to list only missing filenames,

```
constr -f MyFile ArticleName:n
```

should be run.

The number can also be preceded by the keywords `def`, `sch`, `exreg`, `funcreg`, or `condreg` (definition, scheme, and all three types of cluster registrations) to obtain the list of all constructors for virtually any notion. Note that since the three latter do not have their own numbering scheme, you have to count it by yourself, or alternatively invoke the above command without the number (in this case the list of the constructors needed to understand all objects of given type will be obtained).

It is often the case that running this command can help also to understand why the ANALYZER returns `*103` errors although it seems that appropriate environment directives are already there.

4.6 Enhancers

Every Mizar distribution is equipped with a number of programs which help the author to improve the quality of proofs; although they are not simple pretty printers. All this programs suggest possible improvements on the Mizar text.

The calling sequence of the utilities listed below matters; they can be run from Emacs as 'Execute all irrelevant utils'. Otherwise, they can be run from the command line with the `revf` macro (errors flags and their explanations are added to the text) as

```
revf <enhancer> article
```

RELPREM. Searching for irrelevant premises is the most unproblematic and the most popular control which can be performed when writing a Mizar article. RELPREM checks which references are not needed to accept the justification of a sentence; it marks both linking and straightforward references. As it matches also plain VERIFIER errors (besides the schemes), it is useful to use it for verification of the article, although it significantly extends the time of checking in comparison to the standard VERIFIER.

In the case of multiple RELPREM errors reported for a single justification, it means that at least one of these marked references can be removed. Errors `*602` can override themselves. Please also be warned that errors `*602` and `*603` virtually meaning the same might occur – if both `then` and the reference will be removed, the inference will not be accepted anymore, as in the example below.

Marked errors: 602, 603.

```
A1: X <> {}; then
    X <> {} by A1;
::>          *603,602
```

RELINFER. The only controversial exception of the reviewing software is the program which points out irrelevant inferences, i.e. unnecessary steps in a proof (so the references may be added to the next step). It can exceptionally shorten proofs but it may also result in poorer readability of the text. For example, some sentences which are important for the proof technically are marked as irrelevant steps, but their removal may force the user to repeat the same library reference (or their combination) instead of a potentially useful lemma; or the removal may be accidental in some sense, that is steps which are crucial for human understanding of the idea of a proof, but are still unnecessary for machine (e.g., unwinding definitions – definitional expansions). Here the tendencies to reduce the complexity of the proof can be misleading.

```
assume that
A: a = b and
B: b = c and
C: c = d;
D: a = c by A, B;
   a = d by C, D;
::>            *604
```

The error `*604` suggests replacing `D` in the list of references by its justification, i.e. labels `A` and `B`. As one can easily see, in such a way any justification can became

a long list of labels, without significant proof steps which will give the reader an impression about main steps of this proof. Similarly, *605 deal with **then** instead of a labeled sentence.

Errors: 604, 605.

RELITERS. Finding irrelevant steps helps to shorten iterative equalities by removing some of the intermediate terms. Usually, it does not affect readability, especially if one removes steps taking into account that longer terms should be removed first (the system does not care about this).

Errors: 746.

TRIVDEMO. Although Mizar proofs are hierarchical, sometimes after the aforementioned transformations nested proofs can be simplified by the program searching for trivial proofs (that may be reduced to a simple justification, i.e. a list of references preceded by the keyword **by**).

This can be done also in the case of a diffuse statement; then the users have to reproduce the thesis by themselves.

Errors: 607.

CHKLAB. Checking for unused labels should be usually performed after completion of (the part of) a proof and the aforementioned programs. Very often removed unused premises are just library references (for definitions and theorems already proved in MML), but sometimes a reference to a local fact is written accidentally. If a labeled sentence is not used in any reference, after the CHKLAB pass such that label is marked as unnecessary. Still though, the sentence can be needed in a proof via simple linking by the next one (the reserved word **then**in such a case).

Errors: 601.

INACC. Inaccessible (unused) parts of the article can be pointed out as items which are neither labeled nor linked (elements of a proof skeleton are not marked as erroneous).

Errors: 610, 611 (marking the beginning and the end of unused block, respectively).

IRRVOC. Checks which identifiers can be removed from the **vocabularies** directive.

Errors: 709.

Note that this can lead to ACCOMMODATOR errors; this utility might cause some files listed in the **notations** directive to be completely cut off (hence error *830) and this forces the user to remove also such unnecessary declarations.

IRRTHS. Checks which identifiers can be removed from the **theorems** and **schemes** directive.

Errors: 706 (theorems), 707 (schemes).

5. MIZAR MATHEMATICAL LIBRARY

5.1 Axiomatics

5.1.1 *File* HIDDEN. This article documents a part of the Mizar axiomatics – it shows how the primitives of set theory are introduced in the Mizar Mathematical Library. The notions defined here are not subject to standard verification, so the Mizar verifier and other utilities may report errors when processing this article.

It introduces a primitive set:

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

and the predicate of equality of objects which is supposed to be reflexive and symmetric:

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

together with a negated version of the above:

```
notation let x, y be set;
  antonym x <> y for x = y;
end;
```

The last introduced primitive is $\in$ which obtains automatically property of asymmetry.

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

5.1.2 *File* TARSKI. This article defines axiomatic foundations of the Tarski-Grothendieck set theory: extensionality axiom

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

axiom of pair in the setting of functor's definition (although the singleton can be derived from the latter, it is also present):

```
definition let y be set;
  func { y } means
:: TARSKI:def 1
  x in it iff x = y;
end;
```

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

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

```
  reflexivity;
end;
```

Then two other axioms follow: the axiom of union (again in the constructive form of definition) and regularity:

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

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

The Fraenkel scheme playing a role of axiom schema of replacement in ZFC:

```
scheme :: TARSKI:sch 1
  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;
```

The Kuratowski ordered pair definition can be justified in a standard way.

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

and Tarski's Axiom A which implies axioms of power set, infinity, choice and the existence of inaccessible cardinals:

> Axiom A. For every set N there exists a system M of sets which satisfies the following conditions:
> (i) $N \in M$
> (ii) if $X \in M$ and $Y \subseteq X$, then $Y \in M$
> (iii) if $X \in M$ and Z is the system of all subsets of X, then $Z \in M$
> (iv) if $X \subseteq M$ and X and M do not have the same potency, then $X \in M$.
> (as taken from: Alfred Tarski, On well-ordered subsets of any set, *Fundamenta Mathematicae,* vol. 32 (1939), pp. 176–183.)

```
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);
```

where the equipotency was defined as a Mizar predicate:

```
definition let X, Y be set;
  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;
```

5.2 Contents

The MML is roughly divided into five parts (reflected in the `mml.txt` file in the distribution):

—axiomatics, as described in the previous section,
—addenda, in which some articles of technical character are stored (e.g. construction of real numbers via Dedekind cuts), usually authored by the Library Committee,
—EMM (Encyclopedia of Mathematics in Mizar) – files with carefully chosen collection of notions and theorems about a concrete topic (as of now eleven files, mainly on boolean properties of sets, properties of reals, extended real numbers and complex numbers),
—requirements files – short files where **requirements** are proved in a standard way, with automation they provide suspended,
—regular articles.

Although the latter part of MML is not really divided into visible blocks, certain paths which are better developed than others can be identified. Here we can point out set theory, general topology, lattice theory (including continuous lattices), functional analysis and measure theory.

As of now, also 53 facts from the Wiedijk's "Top 100 Mathematical Theorems" are formalized[1], such as Fundamental Theorem of Algebra, Euler's Polyhedron Formula, Ramsey's Theorem, Sylow's theorems, and Brouwer Fixed Point Theorem. Among other important theorems, one can find in MML Jordan Curve Theorem, Birkhoff Variety Theorem, Bertrand's Postulate, Jonsson's theorems for lattices and modular lattices, Nagata-Smirnov Theorem, König's Lemma, Alexander's Lemma, Hahn-Banach Theorem, and Wedderburn Theorem.

The Library Committee of the Association of Mizar Users is in charge of revisions, it also decides about acceptance of new articles (upon suggestions of referees) and about new items in the MML, e.g. EMM files, *Addenda* section.

5.3 Submission of articles

The main effort of the Mizar community is currently developing and maintaining MML. The library is based on the work of more than 230 authors who contributed their articles. Although, in principle, anyone could start developing their own library based on MML,in practice most authors use a local library only to develop

[1] See `http://www.cs.ru.nl/~freek/100/`.

a series of inter-connected articles and as soon as the formalized theory is complete they submit their articles to the centralized MML. This way they can take advantage of the work of the Library Committee that maintains the library, revises its parts to improve the general quality and integrity, and updates the articles whenever there is a change in the Mizar software or the grammar of the Mizar language. Preserving compatibility with MML and keeping track of software changes while maintaining a separate database would be a rather difficult task.

Before submitting an article to MML, the authors should check if it complies with the following criteria:

(1) An article considered for inclusion in MML should contain a significant amount of formalized mathematics in it. In practice that means that it cannot be too short. The recommended length of articles is between 1200 and 5000 lines, with the average length of current MML articles about 2300 lines. As a rule, articles shorter than 1000 lines may be rejected. The articles are required to preserve the restriction of no more than 80 characters per line.

(2) The filenames of MML articles are used in the environment part of other articles for importing selected resources. Therefore, although the system can support a wider class of filenames, the articles submitted to MML are required to use a standardized "8+3" filename format with the obligatory ".miz" extension. The filename must contain only letters, digits and the underscore symbol "_". The first character must be a letter (excluding "x" reserved for a special kind of *encyclopedic* articles. The length of the filename should be between 5 and 8 characters, but 8 characters are preferred. The filename has to be unique, i.e. it must be different from all article names already available in MML. The filename should be an abbreviation corresponding to the title (and contents) of the article.

(3) An article submitted to MML should not rely on a local database (contained in the `prel` subdirectory) built from articles not present in MML. If a local library is used, the preliminaries to the article should be submitted, too.

(4) If the article requires a private vocabulary file (see Section 4.3), it should also be included in the submission. The vocabulary filename should be the same as the article name, except for the obligatory ".voc" extension. The CHECKVOC utility should be used to detect any repeated MML symbols (see Section 4.5). New symbols should not contain extended ASCII characters. One-letter symbols and whitespaces in symbols are not allowed, either. The authors should also make sure that the vocabulary file does not contain any unused symbols.

(5) Obviously, a submitted article should not raise any verification errors. The authors should make sure that they first remove any `@proof`'s they used while writing the article and then call the Mizar VERIFIER. The most recent official Mizar version available should be used.
The article should also be *clean* with respect to standard utilities available in the Mizar distribution: RELPREM, RELINFER, CHKLAB, INACC, TRIVDEMO and RELITERS (see Section 4.6). Unnecessary directives should be removed from the environment part of the article using the IRRVOC and IRRTHS tools.

(6) The submitted article should be accompanied with a bibliographic note and a summary contained in a `*.bib` file (in the Mizar `doc` subdirectory one can find

an example of such a file: `example.bib`). The name of this file has to be same as the name of the corresponding Mizar article. The authors are encouraged to cite external sources they use to carry out the formalization (i.e. non-Mizar) as showed in the `example.bib` file.
Please note that the contents of the file must be in English, since it is used in the process of translating the Mizar article into its natural language representation in the Formalized Mathematics journal, see Section 5.4.

(7) The submission is finally complete when authors fill in a submission form which one can find in the Mizar `doc` subdirectory (`mmldecl.txt` in case of one author and `mmldecls.txt` if there are more) and send it by standard mail or by fax to the following address:

Association of Mizar Users
University of Bialystok
Institute of Mathematics
ul. Akademicka 2
15-267 Bialystok
Poland
Fax: +48-85-745-75-45

PDF versions of these forms are also available for download at the Mizar website.

(8) The contributed files should be attached as a ZIP archive to an e-mail sent to `mml@mizar.uwb.edu.pl`.

5.4 Formalized Mathematics

The benefit that Mizar authors have from submitting their articles to MML is twofold.

As stated in the previous section, if an article gets accepted to MML, the Library Committee starts maintaining it, so that it is always going to be compatible with new versions of the Mizar software and the rest of MML articles. The extra advantage is that the exportable part of the article is then automatically processed by translating software. The result of the translation is published in the journal *Formalized Mathematics*, issued quarterly in both printed and electronic format[2].

Of course all Mizar articles considered for inclusion in MML are checked for correctness by the Mizar system, so they must be logically sound. To guarantee a high quality of their contents, all papers are reviewed by at least three experts in the relevant field.

The system for automatic translation to English and typesetting with LaTeX is designed and implemented by Grzegorz Bancerek (the design is based on previous works of Andrzej Trybulec and Czesław Byliński). The authors can check how their article will be automatically LaTeX-ed if it becomes accepted for publication in *Formalized Mathematics* using the on-line proof-reading system available at `fm.uwb.edu.pl/proof-read/`. The Library Committee strongly encourages authors to use the previewing process.

[2]See the *Formalized Mathematics* website at `fm.mizar.org`.

6. MORE INFORMATION ON MIZAR

This paper was intended to serve as a practical reference manual for basic Mizar terminology. The readers should now be be ready to understand all Mizar texts available in MML and start individual experiments with writing and verifying their own proofs using Mizar. However, Mizar is a complex system and the syntax of its language is very rich, so no doubt there are still many aspects that may yield questions and problems not addressed in this paper. The users are, therefore, advised to search for answers in the Mizar Forum mailing list (`mizar.org/forum`) which is devoted to all aspects of Mizar. There is also a dedicated e-mail helpline, the Mizar User Service (`mus@mizar.uwb.edu.pl`), where one may ask specific questions and make find an expert's help. A great source of useful information is provided by the Mizar Wiki (`wiki.mizar.org`) collaboration tool, where the Mizar community may share information, expertise and ideas.

Finally let us mention here several systems closely related to Mizar that were not covered in the current paper:

—MML Query (a powerful semantic search engine for MML)
`wiki.mizar.org`

—MoMM(a matching, interreduction and database tool for mathematical databases optimized for Mizar)
`wiki.mizar.org/twiki/bin/view/Mizar/MoMM`

—Mizar Proof Advisor
`wiki.mizar.org/twiki/bin/view/Mizar/MizarProofAdvisor`

—MpTP (Mizar Problems for Theorem Proving)
`wiki.mizar.org/twiki/bin/view/Mizar/MpTP`

—Mizar mode for Emacs
`wiki.mizar.org/twiki/bin/view/Mizar/MizarMode`

APPENDIX

A. SKELETONS

A.1 Definitions

A.1.1 *Predicates*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 assume Ψ(x_1, x_2, ..., x_n);
 pred π(x_1, x_2, ..., x_n) means :ident:
  Φ(x_1, x_2, ..., x_n);
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 assume Ψ(x_1, x_2, ..., x_n);
 pred π(x_1, x_2, ..., x_n) means :ident:
  Φ_1(x_1, x_2, ..., x_n) if Γ_1(x_1, x_2, ..., x_n),
  Φ_2(x_1, x_2, ..., x_n) if Γ_2(x_1, x_2, ..., x_n),
  Φ_3(x_1, x_2, ..., x_n) if Γ_3(x_1, x_2, ..., x_n)
  otherwise Φ_n(x_1, x_2, ..., x_n);
 consistency
 proof
  thus Γ_1(x_1, x_2, ..., x_n) & Γ_2(x_1, x_2, ..., x_n) implies
   (Φ_1(x_1, x_2, ..., x_n) iff Φ_2(x_1, x_2, ..., x_n));
  thus Γ_1(x_1, x_2, ..., x_n) & Γ_3(x_1, x_2, ..., x_n) implies
   (Φ_1(x_1, x_2, ..., x_n) iff Φ_3(x_1, x_2, ..., x_n));
  thus Γ_2(x_1, x_2, ..., x_n) & Γ_3(x_1, x_2, ..., x_n) implies
   (Φ_2(x_1, x_2, ..., x_n) iff Φ_3(x_1, x_2, ..., x_n));
 end;
end;
```

A.1.2 *Modes*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 assume Ψ(x_1, x_2, ..., x_n);
 mode μ of x_1, x_2, ..., x_n -> Θ means :ident:
  Φ(x_1, x_2, ..., x_n, it);
 existence
 proof
  thus ex a being Θ st Φ(x_1, x_2, ..., x_n, a);
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 assume Ψ(x_1, x_2, ..., x_n);
 mode μ of x_1, x_2, ..., x_n -> Θ means :ident:
  Φ_1(x_1, x_2, ..., x_n, it) if Γ_1(x_1, x_2, ..., x_n),
```

```
  Φ2(x1, x2, ..., xn, it) if Γ2(x1, x2, ..., xn),
  Φ3(x1, x2, ..., xn, it) if Γ3(x1, x2, ..., xn)
  otherwise Φn(x1, x2, ..., xn, it);
 existence
 proof
  thus Γ1(x1, x2, ..., xn) implies ex a being Θ st Φ1(x1, x2, ..., xn, a);
  thus Γ2(x1, x2, ..., xn) implies ex a being Θ st Φ2(x1, x2, ..., xn, a);
  thus Γ3(x1, x2, ..., xn) implies ex a being Θ st Φ3(x1, x2, ..., xn, a);
  thus not Γ1(x1, x2, ..., xn) & not Γ2(x1, x2, ..., xn) & not Γ3(x1, x2, ..., xn)
   implies ex a being Θ st Φn(x1, x2, ..., xn, a);
 end;
 consistency
 proof
  let a be Θ;
  thus Γ1(x1, x2, ..., xn) & Γ2(x1, x2, ..., xn) implies
   (Φ1(x1, x2, ..., xn, a) iff Φ2(x1, x2, ..., xn, a));
  thus Γ1(x1, x2, ..., xn) & Γ3(x1, x2, ..., xn) implies
   (Φ1(x1, x2, ..., xn, a) iff Φ3(x1, x2, ..., xn, a));
  thus Γ2(x1, x2, ..., xn) & Γ3(x1, x2, ..., xn) implies
   (Φ2(x1, x2, ..., xn, a) iff Φ3(x1, x2, ..., xn, a));
 end;
end;
```

A.1.3 *Functors (means, equals)*

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 assume Ψ(x1, x2, ..., xn);
 func ⊗(x1, x2, ..., xn) -> Θ means :ident:
  Φ(x1, x2, ..., xn, it);
 existence
 proof
  thus ex a being Θ st Φ(x1, x2, ..., xn, a);
 end;
 uniqueness
 proof
  thus for a,b being Θ st Φ(x1, x2, ..., xn, a) & Φ(x1, x2, ..., xn, b)
   holds a = b;
 end;
end;
```

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 assume Ψ(x1, x2, ..., xn);
 func ⊗(x1, x2, ..., xn) -> Θ means :ident:
  Φ1(x1, x2, ..., xn, it) if Γ1(x1, x2, ..., xn),
  Φ2(x1, x2, ..., xn, it) if Γ2(x1, x2, ..., xn),
  Φ3(x1, x2, ..., xn, it) if Γ3(x1, x2, ..., xn)
  otherwise Φn(x1, x2, ..., xn, it);
```

```
 existence
 proof
  thus Γ1(x1,x2,...,xn) implies ex a being Θ st Φ1(x1,x2,...,xn,a);
  thus Γ2(x1,x2,...,xn) implies ex a being Θ st Φ2(x1,x2,...,xn,a);
  thus Γ3(x1,x2,...,xn) implies ex a being Θ st Φ3(x1,x2,...,xn,a);
  thus not Γ1(x1,x2,...,xn) & not Γ2(x1,x2,...,xn) & not Γ3(x1,x2,...,xn)
   implies ex a being Θ st Φn(x1,x2,...,xn,a);
 end;
 uniqueness
 proof
  let a,b be Θ;
  thus Γ1(x1,x2,...,xn) & Φ1(x1,x2,...,xn,a) & Φ1(x1,x2,...,xn,b)
   implies a = b;
  thus Γ2(x1,x2,...,xn) & Φ2(x1,x2,...,xn,a) & Φ2(x1,x2,...,xn,b)
   implies a = b;
  thus Γ3(x1,x2,...,xn) & Φ3(x1,x2,...,xn,a) & Φ3(x1,x2,...,xn,b)
   implies a = b;
  thus not Γ1(x1,x2,...,xn) & not Γ2(x1,x2,...,xn) & not Γ3(x1,x2,...,xn)
   & Φn(x1,x2,...,xn,a) & Φn(x1,x2,...,xn,b) implies a = b;
 end;
 consistency
 proof
  let a be Θ;
  thus Γ1(x1,x2,...,xn) & Γ2(x1,x2,...,xn) implies
   (Φ1(x1,x2,...,xn,a) iff Φ2(x1,x2,...,xn,a));
  thus Γ1(x1,x2,...,xn) & Γ3(x1,x2,...,xn) implies
   (Φ1(x1,x2,...,xn,a) iff Φ3(x1,x2,...,xn,a));
  thus Γ2(x1,x2,...,xn) & Γ3(x1,x2,...,xn) implies
   (Φ2(x1,x2,...,xn,a) iff Φ3(x1,x2,...,xn,a));
 end;
end;

definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 assume Ψ(x1,x2,...,xn);
 func ⊗(x1,x2,...,xn) -> Θ equals :ident:
  τ(x1,x2,...,xn);
 coherence
 proof
  thus τ(x1,x2,...,xn) is Θ;
 end;
end;

definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 assume Ψ(x1,x2,...,xn);
 func ⊗(x1,x2,...,xn) -> Θ equals :ident:
  τ1(x1,x2,...,xn) if Γ1(x1,x2,...,xn),
```

```
  τ2(x1, x2, ..., xn) if Γ2(x1, x2, ..., xn),
  τ3(x1, x2, ..., xn) if Γ3(x1, x2, ..., xn)
  otherwise τn(x1, x2, ..., xn);
 coherence
 proof
  thus Γ1(x1, x2, ..., xn) implies τ1(x1, x2, ..., xn) is Θ;
  thus Γ2(x1, x2, ..., xn) implies τ2(x1, x2, ..., xn) is Θ;
  thus Γ3(x1, x2, ..., xn) implies τ3(x1, x2, ..., xn) is Θ;
  thus not Γ1(x1, x2, ..., xn) & not Γ2(x1, x2, ..., xn) & not Γ3(x1, x2, ..., xn)
   implies τn(x1, x2, ..., xn) is Θ;
 end;
 consistency
 proof
  let a be Θ;
  thus Γ1(x1, x2, ..., xn) & Γ2(x1, x2, ..., xn) implies
   (a = τ1(x1, x2, ..., xn) iff a = τ2(x1, x2, ..., xn));
  thus Γ1(x1, x2, ..., xn) & Γ3(x1, x2, ..., xn) implies
   (a = τ1(x1, x2, ..., xn) iff a = τ3(x1, x2, ..., xn));
  thus Γ2(x1, x2, ..., xn) & Γ3(x1, x2, ..., xn) implies
   (a = τ2(x1, x2, ..., xn) iff a = τ3(x1, x2, ..., xn));
 end;
end;
```

A.1.4 *Attributes*

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 assume Ψ(x1, x2, ..., xn);
 attr xn is α means :ident:
  Φ(x1, x2, ..., xn);
end;

definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 assume Ψ(x1, x2, ..., xn);
 attr xn is α means :ident:
  Φ1(x1, x2, ..., xn) if Γ1(x1, x2, ..., xn),
  Φ2(x1, x2, ..., xn) if Γ2(x1, x2, ..., xn),
  Φ3(x1, x2, ..., xn) if Γ3(x1, x2, ..., xn)
  otherwise Φn(x1, x2, ..., xn);
 consistency
 proof
  thus Γ1(x1, x2, ..., xn) & Γ2(x1, x2, ..., xn) implies
   (Φ1(x1, x2, ..., xn) iff Φ2(x1, x2, ..., xn));
  thus Γ1(x1, x2, ..., xn) & Γ3(x1, x2, ..., xn) implies
   (Φ1(x1, x2, ..., xn) iff Φ3(x1, x2, ..., xn));
  thus Γ2(x1, x2, ..., xn) & Γ3(x1, x2, ..., xn) implies
   (Φ2(x1, x2, ..., xn) iff Φ3(x1, x2, ..., xn));
 end;
```

```
end;
```

A.2 Redefinitions – result type is being changed

A.2.1 *Modes*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 redefine mode μ of x_1,x_2,...,x_n -> Θ;
 coherence
 proof
  thus for a being μ of x_1,x_2,...,x_n holds a is Θ;
 end;
end;
```

A.2.2 *Functors*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 redefine func ⊗(x_1,x_2,...,x_n) -> Θ;
 coherence
 proof
  thus ⊗(x_1,x_2,...,x_n) is Θ;
 end;
end;
```

A.3 Redefinitions – definiens is being changed

A.3.1 *Predicates*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 assume Ψ(x_1,x_2,...,x_n);
 redefine pred π(x_1,x_2,...,x_n) means :ident:
  Φ(x_1,x_2,...,x_n);
 compatibility
 proof
  thus π(x_1,x_2,...,x_n) iff Φ(x_1,x_2,...,x_n);
 end;
end;
```

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 assume Ψ(x_1,x_2,...,x_n);
 redefine pred π(x_1,x_2,...,x_n) means :ident:
  Φ_1(x_1,x_2,...,x_n) if Γ_1(x_1,x_2,...,x_n),
  Φ_2(x_1,x_2,...,x_n) if Γ_2(x_1,x_2,...,x_n),
  Φ_3(x_1,x_2,...,x_n) if Γ_3(x_1,x_2,...,x_n)
  otherwise Φ_n(x_1,x_2,...,x_n);
 consistency;
 compatibility
 proof
```

```
  thus Γ1(x1, x2, ..., xn) implies (π(x1, x2, ..., xn) iff Φ1(x1, x2, ..., xn));
  thus Γ2(x1, x2, ..., xn) implies (π(x1, x2, ..., xn) iff Φ2(x1, x2, ..., xn));
  thus Γ3(x1, x2, ..., xn) implies (π(x1, x2, ..., xn) iff Φ3(x1, x2, ..., xn));
  thus not Γ1(x1, x2, ..., xn) & not Γ2(x1, x2, ..., xn) & not Γ3(x1, x2, ..., xn)
   implies (π(x1, x2, ..., xn) iff Φn(x1, x2, ..., xn));
 end;
end;
```

A.3.2 *Modes*

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 assume Ψ(x1, x2, ..., xn);
 redefine mode μ of x1, x2, ..., xn means :ident:
  Φ(x1, x2, ..., xn, it);
 compatibility
 proof
  thus for a being set holds
   a is μ of x1, x2, ..., xn iff Φ(x1, x2, ..., xn, a);
 end;
end;

definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 assume Ψ(x1, x2, ..., xn);
 redefine mode μ of x1, x2, ..., xn means :ident:
  Φ1(x1, x2, ..., xn, it) if Γ1(x1, x2, ..., xn),
  Φ2(x1, x2, ..., xn, it) if Γ2(x1, x2, ..., xn),
  Φ3(x1, x2, ..., xn, it) if Γ3(x1, x2, ..., xn)
  otherwise Φn(x1, x2, ..., xn, it);
 compatibility
 proof
  let a be set;
  thus Γ1(x1, x2, ..., xn) implies
   (a is μ of x1, x2, ..., xn iff Φ1(x1, x2, ..., xn, a));
  thus Γ2(x1, x2, ..., xn) implies
   (a is μ of x1, x2, ..., xn iff Φ2(x1, x2, ..., xn, a));
  thus Γ3(x1, x2, ..., xn) implies
   (a is μ of x1, x2, ..., xn iff Φ3(x1, x2, ..., xn, a));
  thus not Γ1(x1, x2, ..., xn) & not Γ2(x1, x2, ..., xn) & not Γ3(x1, x2, ..., xn)
   implies (a is μ of x1, x2, ..., xn iff Φn(x1, x2, ..., xn, a));
 end;
 consistency;
end;
```

A.3.3 *Functors (means, equals)*

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 assume Ψ(x1, x2, ..., xn);
```

```
 redefine func ⊗(x_1, x_2, ..., x_n) means :ident:
  Φ(x_1, x_2, ..., x_n, it);
 compatibility
 proof
  thus for a being Θ holds a = ⊗(x_1, x_2, ..., x_n) iff Φ(x_1, x_2, ..., x_n, a);
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 assume Ψ(x_1, x_2, ..., x_n);
 redefine func ⊗(x_1, x_2, ..., x_n) means :ident:
  Φ_1(x_1, x_2, ..., x_n, it) if Γ_1(x_1, x_2, ..., x_n),
  Φ_2(x_1, x_2, ..., x_n, it) if Γ_2(x_1, x_2, ..., x_n),
  Φ_3(x_1, x_2, ..., x_n, it) if Γ_3(x_1, x_2, ..., x_n)
  otherwise Φ_n(x_1, x_2, ..., x_n, it);
 consistency;
 compatibility
 proof
  let a be Θ;
  thus Γ_1(x_1, x_2, ..., x_n) implies
   (a = ⊗(x_1, x_2, ..., x_n) iff Φ_1(x_1, x_2, ..., x_n, a));
  thus Γ_2(x_1, x_2, ..., x_n) implies
   (a = ⊗(x_1, x_2, ..., x_n) iff Φ_2(x_1, x_2, ..., x_n, a));
  thus Γ_3(x_1, x_2, ..., x_n) implies
   (a = ⊗(x_1, x_2, ..., x_n) iff Φ_3(x_1, x_2, ..., x_n, a));
  thus not Γ_1(x_1, x_2, ..., x_n) & not Γ_2(x_1, x_2, ..., x_n) & not Γ_3(x_1, x_2, ..., x_n)
   implies (a = ⊗(x_1, x_2, ..., x_n) iff Φ_3(x_1, x_2, ..., x_n, a));
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 assume Ψ(x_1, x_2, ..., x_n);
 redefine func ⊗(x_1, x_2, ..., x_n) equals :ident:
  τ(x_1, x_2, ..., x_n);
 compatibility
 proof
  thus for a being Θ holds a = ⊗(x_1, x_2, ..., x_n) iff a = τ(x_1, x_2, ..., x_n);
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 assume Ψ(x_1, x_2, ..., x_n);
 redefine func ⊗(x_1, x_2, ..., x_n) equals :ident:
  τ_1(x_1, x_2, ..., x_n) if Γ_1(x_1, x_2, ..., x_n),
  τ_2(x_1, x_2, ..., x_n) if Γ_2(x_1, x_2, ..., x_n),
  τ_3(x_1, x_2, ..., x_n) if Γ_3(x_1, x_2, ..., x_n)
```

```
  otherwise τ_n(x_1, x_2, ..., x_n);
 consistency;
 compatibility
 proof
  let a be Θ;
  thus Γ_1(x_1, x_2, ..., x_n) implies
   (a = ⊗(x_1, x_2, ..., x_n) iff a = τ_1(x_1, x_2, ..., x_n));
  thus Γ_2(x_1, x_2, ..., x_n) implies
   (a = ⊗(x_1, x_2, ..., x_n) iff a = τ_2(x_1, x_2, ..., x_n));
  thus Γ_3(x_1, x_2, ..., x_n) implies
   (a = ⊗(x_1, x_2, ..., x_n) iff a = τ_3(x_1, x_2, ..., x_n));
  thus not Γ_1(x_1, x_2, ..., x_n) & not Γ_2(x_1, x_2, ..., x_n) & not Γ_3(x_1, x_2, ..., x_n)
   implies (a = ⊗(x_1, x_2, ..., x_n) iff a = τ_n(x_1, x_2, ..., x_n));
 end;
end;
```

A.3.4 *Attributes*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 assume Ψ(x_1, x_2, ..., x_n);
 redefine attr x_n is α means :ident:
  Φ(x_1, x_2, ..., x_n);
 compatibility
 proof
  thus x_n is α iff Φ(x_1, x_2, ..., x_n);
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 assume Ψ(x_1, x_2, ..., x_n);
 redefine attr x_n is α means :ident:
  Φ_1(x_1, x_2, ..., x_n) if Γ_1(x_1, x_2, ..., x_n),
  Φ_2(x_1, x_2, ..., x_n) if Γ_2(x_1, x_2, ..., x_n),
  Φ_3(x_1, x_2, ..., x_n) if Γ_3(x_1, x_2, ..., x_n)
  otherwise Φ_n(x_1, x_2, ..., x_n);
 consistency;
 compatibility
 proof
  thus Γ_1(x_1, x_2, ..., x_n) implies (x_n is α iff Φ_1(x_1, x_2, ..., x_n));
  thus Γ_2(x_1, x_2, ..., x_n) implies (x_n is α iff Φ_2(x_1, x_2, ..., x_n));
  thus Γ_3(x_1, x_2, ..., x_n) implies (x_n is α iff Φ_3(x_1, x_2, ..., x_n));
  thus not Γ_1(x_1, x_2, ..., x_n) & not Γ_2(x_1, x_2, ..., x_n) &
   not Γ_3(x_1, x_2, ..., x_n) implies (x_n is α iff Φ_n(x_1, x_2, ..., x_n));
 end;
end;
```

A.4 Registrations

A.4.1 *Existential*

```
registration
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 cluster α_1 ... α_m Θ;
 existence
 proof
  thus ex a being Θ st a is α_1 ... α_m;
 end;
end;
```

A.4.2 *Conditional*

```
registration
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 cluster α_1 ... α_m -> α_{m+1} ... α_{m+1+k} Θ;
 coherence
 proof
  thus for a being Θ st a is α_1 ... α_m holds a is α_{m+1} ... α_{m+1+k};
 end;
end;
```

```
registration
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 cluster -> α_1 ... α_m Θ;
 coherence
 proof
  thus for a being Θ holds a is α_1 ... α_m;
 end;
end;
```

A.4.3 *Functorial*

```
registration
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 cluster τ(x_1, x_2, ..., x_n) -> α_1 ... α_m;
 coherence
 proof
  thus τ(x_1, x_2, ..., x_n) is α_1 ... α_m;
 end;
end;
```

```
registration
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 cluster τ(x_1, x_2, ..., x_n) -> α_1 ... α_m Θ;
 coherence
 proof
  thus for a being Θ st a = τ(x_1, x_2, ..., x_n) holds a is α_1 ... α_m;
 end;
end;
```

A.5 Properties in definitions

A.5.1 *Predicates.*

A.5.1.1 *Reflexivity*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1,y_2 be Θ_{n+1};
 pred π(y_1,y_2) means :ident:
  Φ(x_1,x_2,...,x_n,y_1,y_2);
 reflexivity
 proof
  thus for a being Θ_{n+1} holds Φ(x_1,x_2,...,x_n,a,a);
 end;
end;
```

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1,y_2 be Θ_{n+1};
 pred π(y_1,y_2) means :ident:
  Φ_1(x_1,x_2,...,x_n,y_1,y_2) if Γ_1(x_1,x_2,...,x_n,y_1,y_2),
  Φ_2(x_1,x_2,...,x_n,y_1,y_2) if Γ_2(x_1,x_2,...,x_n,y_1,y_2),
  Φ_3(x_1,x_2,...,x_n,y_1,y_2) if Γ_3(x_1,x_2,...,x_n,y_1,y_2)
  otherwise Φ_n(x_1,x_2,...,x_n,y_1,y_2);
 consistency;
 reflexivity
 proof
  thus for a being Θ_{n+1} holds
   (Γ_1(x_1,x_2,...,x_n,a,a) implies Φ_1(x_1,x_2,...,x_n,a,a)) &
   (Γ_2(x_1,x_2,...,x_n,a,a) implies Φ_2(x_1,x_2,...,x_n,a,a)) &
   (Γ_3(x_1,x_2,...,x_n,a,a) implies Φ_3(x_1,x_2,...,x_n,a,a)) &
   (not Γ_1(x_1,x_2,...,x_n,a,a) & not Γ_2(x_1,x_2,...,x_n,a,a) &
    not Γ_3(x_1,x_2,...,x_n,a,a) implies Φ_n(x_1,x_2,...,x_n,a,a));
 end;
end;
```

A.5.1.2 *Irreflexivity*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1,y_2 be Θ_{n+1};
 pred π(y_1,y_2) means :ident:
  Φ(x_1,x_2,...,x_n,y_1,y_2);
 irreflexivity
 proof
  thus for a being Θ_{n+1} holds not Φ(x_1,x_2,...,x_n,a,a);
 end;
end;
```

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1,y_2 be Θ_{n+1};
 pred π(y_1,y_2) means :ident:
  Φ_1(x_1,x_2,...,x_n,y_1,y_2) if Γ_1(x_1,x_2,...,x_n,y_1,y_2),
```

```
  Φ2(x1, x2, ..., xn, y1, y2) if Γ2(x1, x2, ..., xn, y1, y2),
  Φ3(x1, x2, ..., xn, y1, y2) if Γ3(x1, x2, ..., xn, y1, y2)
  otherwise Φn(x1, x2, ..., xn, y1, y2);
 consistency;
 irreflexivity
 proof
  thus for a being Θn+1 holds not
   (
    (Γ1(x1, x2, ..., xn, a, a) implies Φ1(x1, x2, ..., xn, a, a)) &
    (Γ2(x1, x2, ..., xn, a, a) implies Φ2(x1, x2, ..., xn, a, a)) &
    (Γ3(x1, x2, ..., xn, a, a) implies Φ3(x1, x2, ..., xn, a, a)) &
    (not Γ1(x1, x2, ..., xn, a, a) & not Γ2(x1, x2, ..., xn, a, a) &
     not Γ3(x1, x2, ..., xn, a, a) implies Φn(x1, x2, ..., xn, a, a))
   );
 end;
end;
```

A.5.1.3 *Symmetry*

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn, y1, y2 be Θn+1;
 pred π(y1, y2) means :ident:
  Φ(x1, x2, ..., xn, y1, y2);
 symmetry
 proof
  thus for a, b being Θn+1 st Φ(x1, x2, ..., xn, a, b) holds
   Φ(x1, x2, ..., xn, b, a);
 end;
end;

definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn, y1, y2 be Θn+1;
 pred π(y1, y2) means :ident:
  Φ1(x1, x2, ..., xn, y1, y2) if Γ1(x1, x2, ..., xn, y1, y2),
  Φ2(x1, x2, ..., xn, y1, y2) if Γ2(x1, x2, ..., xn, y1, y2),
  Φ3(x1, x2, ..., xn, y1, y2) if Γ3(x1, x2, ..., xn, y1, y2)
  otherwise Φn(x1, x2, ..., xn, y1, y2);
 consistency;
 symmetry
 proof
  thus for a, b being Θn+1 st
   (
    (Γ1(x1, x2, ..., xn, a, b) implies Φ1(x1, x2, ..., xn, a, b)) &
    (Γ2(x1, x2, ..., xn, a, b) implies Φ2(x1, x2, ..., xn, a, b)) &
    (Γ3(x1, x2, ..., xn, a, b) implies Φ3(x1, x2, ..., xn, a, b)) &
    (not Γ1(x1, x2, ..., xn, a, b) & not Γ2(x1, x2, ..., xn, a, b) &
     not Γ3(x1, x2, ..., xn, a, b) implies Φn(x1, x2, ..., xn, a, b))
   ) holds
   (
```

```
    (Γ1(x1, x2, ..., xn, b, a) implies Φ1(x1, x2, ..., xn, b, a)) &
    (Γ2(x1, x2, ..., xn, b, a) implies Φ2(x1, x2, ..., xn, b, a)) &
    (Γ3(x1, x2, ..., xn, b, a) implies Φ3(x1, x2, ..., xn, b, a)) &
    (not Γ1(x1, x2, ..., xn, b, a) & not Γ2(x1, x2, ..., xn, b, a) &
     not Γ3(x1, x2, ..., xn, b, a) implies Φn(x1, x2, ..., xn, b, a))
   );
 end;
end;
```

A.5.1.4 *Asymmetry*

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn, y1, y2 be Θn+1;
 pred π(y1, y2) means :ident:
  Φ(x1, x2, ..., xn, y1, y2);
 asymmetry
 proof
  thus for a, b being Θn+1 st Φ(x1, x2, ..., xn, a, b) holds
   not Φ(x1, x2, ..., xn, b, a);
 end;
end;

definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn, y1, y2 be Θn+1;
 pred π(y1, y2) means :ident:
  Φ1(x1, x2, ..., xn, y1, y2) if Γ1(x1, x2, ..., xn, y1, y2),
  Φ2(x1, x2, ..., xn, y1, y2) if Γ2(x1, x2, ..., xn, y1, y2),
  Φ3(x1, x2, ..., xn, y1, y2) if Γ3(x1, x2, ..., xn, y1, y2)
  otherwise Φn(x1, x2, ..., xn, y1, y2);
 consistency;
 asymmetry
 proof
  thus for a, b being Θn+1 st
   (
    (Γ1(x1, x2, ..., xn, a, b) implies Φ1(x1, x2, ..., xn, a, b)) &
    (Γ2(x1, x2, ..., xn, a, b) implies Φ2(x1, x2, ..., xn, a, b)) &
    (Γ3(x1, x2, ..., xn, a, b) implies Φ3(x1, x2, ..., xn, a, b)) &
    (not Γ1(x1, x2, ..., xn, a, b) & not Γ2(x1, x2, ..., xn, a, b) &
     not Γ3(x1, x2, ..., xn, a, b) implies Φn(x1, x2, ..., xn, a, b))
   ) holds not
   (
    (Γ1(x1, x2, ..., xn, b, a) implies Φ1(x1, x2, ..., xn, b, a)) &
    (Γ2(x1, x2, ..., xn, b, a) implies Φ2(x1, x2, ..., xn, b, a)) &
    (Γ3(x1, x2, ..., xn, b, a) implies Φ3(x1, x2, ..., xn, b, a)) &
    (not Γ1(x1, x2, ..., xn, b, a) & not Γ2(x1, x2, ..., xn, b, a) &
     not Γ3(x1, x2, ..., xn, b, a) implies Φn(x1, x2, ..., xn, b, a))
   );
 end;
end;
```

A.5.1.5 *Connectedness*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1,y_2 be Θ_{n+1};
 pred π(y_1,y_2) means :ident:
  Φ(x_1,x_2,...,x_n,y_1,y_2);
 connectedness
 proof
  thus for a, b being Θ_{n+1} holds Φ(x_1,x_2,...,x_n,a,b) or
   Φ(x_1,x_2,...,x_n,b,a);
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1,y_2 be Θ_{n+1};
 pred π(y_1,y_2) means :ident:
  Φ_1(x_1,x_2,...,x_n,y_1,y_2) if Γ_1(x_1,x_2,...,x_n,y_1,y_2),
  Φ_2(x_1,x_2,...,x_n,y_1,y_2) if Γ_2(x_1,x_2,...,x_n,y_1,y_2),
  Φ_3(x_1,x_2,...,x_n,y_1,y_2) if Γ_3(x_1,x_2,...,x_n,y_1,y_2)
  otherwise Φ_n(x_1,x_2,...,x_n,y_1,y_2);
 consistency;
 connectedness
 proof
  thus for a, b being Θ_{n+1} holds
   (
    (Γ_1(x_1,x_2,...,x_n,a,b) implies Φ_1(x_1,x_2,...,x_n,a,b)) &
    (Γ_2(x_1,x_2,...,x_n,a,b) implies Φ_2(x_1,x_2,...,x_n,a,b)) &
    (Γ_3(x_1,x_2,...,x_n,a,b) implies Φ_3(x_1,x_2,...,x_n,a,b)) &
    (not Γ_1(x_1,x_2,...,x_n,a,b) & not Γ_2(x_1,x_2,...,x_n,a,b) &
     not Γ_3(x_1,x_2,...,x_n,a,b) implies Φ_n(x_1,x_2,...,x_n,a,b))
   ) or
   (
    (Γ_1(x_1,x_2,...,x_n,b,a) implies Φ_1(x_1,x_2,...,x_n,b,a)) &
    (Γ_2(x_1,x_2,...,x_n,b,a) implies Φ_2(x_1,x_2,...,x_n,b,a)) &
    (Γ_3(x_1,x_2,...,x_n,b,a) implies Φ_3(x_1,x_2,...,x_n,b,a)) &
    (not Γ_1(x_1,x_2,...,x_n,b,a) & not Γ_2(x_1,x_2,...,x_n,b,a) &
     not Γ_3(x_1,x_2,...,x_n,b,a) implies Φ_n(x_1,x_2,...,x_n,b,a))
   );
 end;
end;
```

A.5.2 *Functors.*

A.5.2.1 *Involutiveness (means, equals)*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 func ⊗(x_n) -> Θ_n means :ident:
  Φ(x_1,x_2,...,x_n,it);
 existence;
```

```
 uniqueness;
 involutiveness
 proof
  thus for a, b being Θ_n st Φ(x_1, x_2, ..., x_{n-1}, b, a)
   holds Φ(x_1, x_2, ..., x_{n-1}, a, b);
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 func ⊗(x_n) -> Θ_n means :ident:
  Φ_1(x_1, x_2, ..., x_n, it) if Γ_1(x_1, x_2, ..., x_n),
  Φ_2(x_1, x_2, ..., x_n, it) if Γ_2(x_1, x_2, ..., x_n),
  Φ_3(x_1, x_2, ..., x_n, it) if Γ_3(x_1, x_2, ..., x_n)
  otherwise Φ_n(x_1, x_2, ..., x_n, it);
 existence;
 uniqueness;
 consistency;
 involutiveness
 proof
  thus for a, b being Θ_n st
   (
    (Γ_1(x_1, x_2, ..., x_{n-1}, b) implies Φ_1(x_1, x_2, ..., x_{n-1}, b, a)) &
    (Γ_2(x_1, x_2, ..., x_{n-1}, b) implies Φ_2(x_1, x_2, ..., x_{n-1}, b, a)) &
    (Γ_3(x_1, x_2, ..., x_{n-1}, b) implies Φ_3(x_1, x_2, ..., x_{n-1}, b, a)) &
    (not Γ_1(x_1, x_2, ..., x_{n-1}, b) & not Γ_2(x_1, x_2, ..., x_{n-1}, b) &
     not Γ_3(x_1, x_2, ..., x_{n-1}, b) implies Φ_n(x_1, x_2, ..., x_{n-1}, b, a))
   ) holds
   (
    (Γ_1(x_1, x_2, ..., x_{n-1}, a) implies Φ_1(x_1, x_2, ..., x_{n-1}, a, b)) &
    (Γ_2(x_1, x_2, ..., x_{n-1}, a) implies Φ_2(x_1, x_2, ..., x_{n-1}, a, b)) &
    (Γ_3(x_1, x_2, ..., x_{n-1}, a) implies Φ_3(x_1, x_2, ..., x_{n-1}, a, b)) &
    (not Γ_1(x_1, x_2, ..., x_{n-1}, a) & not Γ_2(x_1, x_2, ..., x_{n-1}, a) &
     not Γ_3(x_1, x_2, ..., x_{n-1}, a) implies Φ_n(x_1, x_2, ..., x_{n-1}, a, b))
   );
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 func ⊗(x_n) -> Θ_n equals :ident:
  τ(x_1, x_2, ..., x_n);
 coherence;
 involutiveness
 proof
  thus for a, b being Θ_n st a = τ(x_1, x_2, ..., x_{n-1}, b) holds
   b = τ(x_1, x_2, ..., x_{n-1}, a);
 end;
```

```
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 func ⊗(x_n) -> Θ_n equals :ident:
  τ_1(x_1, x_2, ..., x_n) if Γ_1(x_1, x_2, ..., x_n),
  τ_2(x_1, x_2, ..., x_n) if Γ_2(x_1, x_2, ..., x_n),
  τ_3(x_1, x_2, ..., x_n) if Γ_3(x_1, x_2, ..., x_n)
  otherwise τ_n(x_1, x_2, ..., x_n);
 coherence;
 consistency;
 involutiveness
 proof
  for a, b being Θ_n st
   (Γ_1(x_1, x_2, ..., x_{n-1}, b) implies a = τ_1(x_1, x_2, ..., x_{n-1}, b)) &
   (Γ_2(x_1, x_2, ..., x_{n-1}, b) implies a = τ_2(x_1, x_2, ..., x_{n-1}, b)) &
   (Γ_3(x_1, x_2, ..., x_{n-1}, b) implies a = τ_3(x_1, x_2, ..., x_{n-1}, b)) &
   (not Γ_1(x_1, x_2, ..., x_{n-1}, b) & not Γ_2(x_1, x_2, ..., x_{n-1}, b) &
     not Γ_3(x_1, x_2, ..., x_{n-1}, b) implies a = τ_n(x_1, x_2, ..., x_{n-1}, b))
  holds
   (Γ_1(x_1, x_2, ..., x_{n-1}, a) implies b = τ_1(x_1, x_2, ..., x_{n-1}, a)) &
   (Γ_2(x_1, x_2, ..., x_{n-1}, a) implies b = τ_2(x_1, x_2, ..., x_{n-1}, a)) &
   (Γ_3(x_1, x_2, ..., x_{n-1}, a) implies b = τ_3(x_1, x_2, ..., x_{n-1}, a)) &
   (not Γ_1(x_1, x_2, ..., x_{n-1}, a) & not Γ_2(x_1, x_2, ..., x_{n-1}, a) &
     not Γ_3(x_1, x_2, ..., x_{n-1}, a) implies b = τ_n(x_1, x_2, ..., x_{n-1}, a));
 end;
end;
```

A.5.2.2 *Projectivity (means, equals)*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 func ⊗(x_n) -> Θ_{n+1} means :ident:
  Φ(x_1, x_2, ..., x_n, it);
 existence;
 uniqueness;
 projectivity
 proof
  thus for a, b being Θ_{n+1} st Φ(x_1, x_2, ..., x_{n-1}, b, a) holds
   Φ(x_1, x_2, ..., x_{n-1}, a, a);
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 func ⊗(x_n) -> Θ_{n+1} means :ident:
  Φ_1(x_1, x_2, ..., x_n, it) if Γ_1(x_1, x_2, ..., x_n),
  Φ_2(x_1, x_2, ..., x_n, it) if Γ_2(x_1, x_2, ..., x_n),
  Φ_3(x_1, x_2, ..., x_n, it) if Γ_3(x_1, x_2, ..., x_n)
```

```
  otherwise Φ_n(x_1, x_2, ..., x_n, it);
 existence;
 uniqueness;
 consistency;
 projectivity
 proof
  thus for a, b being Θ_{n+1} st
   (
    (Γ_1(x_1, x_2, ..., x_{n-1}, b) implies Φ_1(x_1, x_2, ..., x_{n-1}, b, a)) &
    (Γ_2(x_1, x_2, ..., x_{n-1}, b) implies Φ_2(x_1, x_2, ..., x_{n-1}, b, a)) &
    (Γ_3(x_1, x_2, ..., x_{n-1}, b) implies Φ_3(x_1, x_2, ..., x_{n-1}, b, a)) &
    (not Γ_1(x_1, x_2, ..., x_{n-1}, b) & not Γ_2(x_1, x_2, ..., x_{n-1}, b) &
     not Γ_3(x_1, x_2, ..., x_{n-1}, b) implies Φ_n(x_1, x_2, ..., x_{n-1}, b, a))
   ) holds
   (
    (Γ_1(x_1, x_2, ..., x_{n-1}, a) implies Φ_1(x_1, x_2, ..., x_{n-1}, a, a)) &
    (Γ_2(x_1, x_2, ..., x_{n-1}, a) implies Φ_2(x_1, x_2, ..., x_{n-1}, a, a)) &
    (Γ_3(x_1, x_2, ..., x_{n-1}, a) implies Φ_3(x_1, x_2, ..., x_{n-1}, a, a)) &
    (not Γ_1(x_1, x_2, ..., x_{n-1}, a) & not Γ_2(x_1, x_2, ..., x_{n-1}, a) &
     not Γ_3(x_1, x_2, ..., x_{n-1}, a) implies Φ_n(x_1, x_2, ..., x_{n-1}, a, a))
   );
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 func ⊗(x_n) -> Θ_{n+1} equals :ident:
  τ(x_1, x_2, ..., x_n);
 coherence;
 projectivity
 proof
  thus for a, b being Θ_{n+1} st a = τ(x_1, x_2, ..., x_{n-1}, b) holds
   a = τ(x_1, x_2, ..., x_{n-1}, a);
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 func ⊗(x_n) -> Θ_{n+1} equals :ident:
  τ_1(x_1, x_2, ..., x_n) if Γ_1(x_1, x_2, ..., x_n),
  τ_2(x_1, x_2, ..., x_n) if Γ_2(x_1, x_2, ..., x_n),
  τ_3(x_1, x_2, ..., x_n) if Γ_3(x_1, x_2, ..., x_n)
  otherwise τ_n(x_1, x_2, ..., x_n);
 coherence;
 consistency;
 projectivity
 proof
  thus for a, b being Θ_{n+1} st
```

```
   (
    (Γ1(x1, x2, ..., xn−1, b) implies a = τ1(x1, x2, ..., xn−1, b)) &
    (Γ2(x1, x2, ..., xn−1, b) implies a = τ2(x1, x2, ..., xn−1, b)) &
    (Γ3(x1, x2, ..., xn−1, b) implies a = τ3(x1, x2, ..., xn−1, b)) &
    (not Γ1(x1, x2, ..., xn−1, b) & not Γ2(x1, x2, ..., xn−1, b) &
     not Γ3(x1, x2, ..., xn−1, b) implies a = τn(x1, x2, ..., xn−1, b))
   ) holds
   (
    (Γ1(x1, x2, ..., xn−1, a) implies a = τ1(x1, x2, ..., xn−1, a)) &
    (Γ2(x1, x2, ..., xn−1, a) implies a = τ2(x1, x2, ..., xn−1, a)) &
    (Γ3(x1, x2, ..., xn−1, a) implies a = τ3(x1, x2, ..., xn−1, a)) &
    (not Γ1(x1, x2, ..., xn−1, a) & not Γ2(x1, x2, ..., xn−1, a) &
     not Γ3(x1, x2, ..., xn−1, a) implies a = τn(x1, x2, ..., xn−1, a))
   );
 end;
end;
```

A.5.2.3 *Idempotence (means, equals)*

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn, y1, y2 be Θn+1;
 func ⊗(y1, y2) -> Θn+2 means :ident:
  Φ(x1, x2, ..., xn, y1, y2, it);
 existence;
 uniqueness;
 idempotence
 proof
  thus for a being Θn+1 holds Φ(x1, x2, ..., xn, a, a, a);
 end;
end;

definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn, y1, y2 be Θn+1;
 func ⊗(y1, y2) -> Θn+2 means :ident:
  Φ1(x1, x2, ..., xn, y1, y2, it) if Γ1(x1, x2, ..., xn, y1, y2),
  Φ2(x1, x2, ..., xn, y1, y2, it) if Γ2(x1, x2, ..., xn, y1, y2),
  Φ3(x1, x2, ..., xn, y1, y2, it) if Γ3(x1, x2, ..., xn, y1, y2)
  otherwise Φn(x1, x2, ..., xn, y1, y2, it);
 existence;
 uniqueness;
 consistency;
 idempotence
 proof
  thus for a being Θn+1 holds
   (Γ1(x1, x2, ..., xn, a, a) implies Φ1(x1, x2, ..., xn, a, a, a)) &
   (Γ2(x1, x2, ..., xn, a, a) implies Φ2(x1, x2, ..., xn, a, a, a)) &
   (Γ3(x1, x2, ..., xn, a, a) implies Φ3(x1, x2, ..., xn, a, a, a)) &
   (not Γ1(x1, x2, ..., xn, a, a) & not Γ2(x1, x2, ..., xn, a, a) &
    not Γ3(x1, x2, ..., xn, a, a) implies Φn(x1, x2, ..., xn, a, a, a));
```

```
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1, y_2 be Θ_{n+1};
 func ⊗(y_1, y_2) -> Θ_{n+2} equals :ident:
  τ(x_1, x_2, ..., x_n, y_1, y_2);
 coherence;
 idempotence
 proof
  thus for a being Θ_{n+1} holds a = τ(x_1, x_2, ..., x_n, a, a);
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1, y_2 be Θ_{n+1};
 func ⊗(y_1, y_2) -> Θ_{n+2} equals :ident:
  τ_1(x_1, x_2, ..., x_n, y_1, y_2) if Γ_1(x_1, x_2, ..., x_n, y_1, y_2),
  τ_2(x_1, x_2, ..., x_n, y_1, y_2) if Γ_2(x_1, x_2, ..., x_n, y_1, y_2),
  τ_3(x_1, x_2, ..., x_n, y_1, y_2) if Γ_3(x_1, x_2, ..., x_n, y_1, y_2)
  otherwise τ_n(x_1, x_2, ..., x_n, y_1, y_2);
 coherence;
 consistency;
 idempotence
 proof
  thus for a being Θ_{n+1} holds
   (Γ_1(x_1, x_2, ..., x_n, a, a) implies a = τ_1(x_1, x_2, ..., x_n, a, a)) &
   (Γ_2(x_1, x_2, ..., x_n, a, a) implies a = τ_2(x_1, x_2, ..., x_n, a, a)) &
   (Γ_3(x_1, x_2, ..., x_n, a, a) implies a = τ_3(x_1, x_2, ..., x_n, a, a)) &
   (not Γ_1(x_1, x_2, ..., x_n, a, a) & not Γ_2(x_1, x_2, ..., x_n, a, a) &
    not Γ_3(x_1, x_2, ..., x_n, a, a) implies a = τ_n(x_1, x_2, ..., x_n, a, a));
 end;
end;
```

A.5.2.4 *Commutativity (means, equals)*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1, y_2 be Θ_{n+1};
 func ⊗(y_1, y_2) -> Θ_{n+2} means :ident:
  Φ(x_1, x_2, ..., x_n, y_1, y_2, it);
 existence;
 uniqueness;
 commutativity
 proof
  thus for a being Θ_{n+2}, b, c being Θ_{n+1} st Φ(x_1, x_2, ..., x_n, b, c, a)
   holds Φ(x_1, x_2, ..., x_n, c, b, a);
 end;
end;

definition
```

```
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1,y_2 be Θ_{n+1};
 func ⊗(y_1,y_2) -> Θ_{n+2} means :ident:
  Φ_1(x_1,x_2,...,x_n,y_1,y_2,it) if Γ_1(x_1,x_2,...,x_n,y_1,y_2),
  Φ_2(x_1,x_2,...,x_n,y_1,y_2,it) if Γ_2(x_1,x_2,...,x_n,y_1,y_2),
  Φ_3(x_1,x_2,...,x_n,y_1,y_2,it) if Γ_3(x_1,x_2,...,x_n,y_1,y_2)
  otherwise Φ_n(x_1,x_2,...,x_n,y_1,y_2,it);
 existence;
 uniqueness;
 commutativity
 proof
  thus for a being Θ_{n+2}, b, c being Θ_{n+1} st
   (
    (Γ_1(x_1,x_2,...,x_n,b,c) implies Φ_1(x_1,x_2,...,x_n,b,c,a)) &
    (Γ_2(x_1,x_2,...,x_n,b,c) implies Φ_2(x_1,x_2,...,x_n,b,c,a)) &
    (Γ_3(x_1,x_2,...,x_n,b,c) implies Φ_3(x_1,x_2,...,x_n,b,c,a)) &
    (not Γ_1(x_1,x_2,...,x_n,b,c) & not Γ_2(x_1,x_2,...,x_n,b,c) &
     not Γ_3(x_1,x_2,...,x_n,b,c) implies Φ_n(x_1,x_2,...,x_n,b,c,a))
   ) holds
   (
    (Γ_1(x_1,x_2,...,x_n,c,b) implies Φ_1(x_1,x_2,...,x_n,c,b,a)) &
    (Γ_2(x_1,x_2,...,x_n,c,b) implies Φ_2(x_1,x_2,...,x_n,c,b,a)) &
    (Γ_3(x_1,x_2,...,x_n,c,b) implies Φ_3(x_1,x_2,...,x_n,c,b,a)) &
    (not Γ_1(x_1,x_2,...,x_n,c,b) & not Γ_2(x_1,x_2,...,x_n,c,b) &
     not Γ_3(x_1,x_2,...,x_n,c,b) implies Φ_n(x_1,x_2,...,x_n,c,b,a))
   );
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1,y_2 be Θ_{n+1};
 func ⊗(y_1,y_2) -> Θ_{n+2} equals :ident:
  τ(x_1,x_2,...,x_n,y_1,y_2);
 coherence;
 commutativity
 proof
  thus for a being Θ_{n+2}, b, c being Θ_{n+1} st a = τ(x_1,x_2,...,x_n,b,c)
   holds a = τ(x_1,x_2,...,x_n,c,b);
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1,y_2 be Θ_{n+1};
 func ⊗(y_1,y_2) -> Θ_{n+2} equals :ident:
  τ_1(x_1,x_2,...,x_n,y_1,y_2) if Γ_1(x_1,x_2,...,x_n,y_1,y_2),
  τ_2(x_1,x_2,...,x_n,y_1,y_2) if Γ_2(x_1,x_2,...,x_n,y_1,y_2),
  τ_3(x_1,x_2,...,x_n,y_1,y_2) if Γ_3(x_1,x_2,...,x_n,y_1,y_2)
  otherwise τ_n(x_1,x_2,...,x_n,y_1,y_2);
 coherence;
```

```
 consistency;
 commutativity
 proof
  thus for a being Θ_{n+2}, b, c being Θ_{n+1} st
   (
    (Γ_1(x_1, x_2, ..., x_n, b, c) implies a = τ_1(x_1, x_2, ..., x_n, b, c)) &
    (Γ_2(x_1, x_2, ..., x_n, b, c) implies a = τ_2(x_1, x_2, ..., x_n, b, c)) &
    (Γ_3(x_1, x_2, ..., x_n, b, c) implies a = τ_3(x_1, x_2, ..., x_n, b, c)) &
    (not Γ_1(x_1, x_2, ..., x_n, b, c) & not Γ_2(x_1, x_2, ..., x_n, b, c) &
     not Γ_3(x_1, x_2, ..., x_n, b, c) implies a = τ_n(x_1, x_2, ..., x_n, b, c))
   ) holds
   (
    (Γ_1(x_1, x_2, ..., x_n, c, b) implies a = τ_1(x_1, x_2, ..., x_n, c, b)) &
    (Γ_2(x_1, x_2, ..., x_n, c, b) implies a = τ_2(x_1, x_2, ..., x_n, c, b)) &
    (Γ_3(x_1, x_2, ..., x_n, c, b) implies a = τ_3(x_1, x_2, ..., x_n, c, b)) &
    (not Γ_1(x_1, x_2, ..., x_n, c, b) & not Γ_2(x_1, x_2, ..., x_n, c, b) &
     not Γ_3(x_1, x_2, ..., x_n, c, b) implies a = τ_n(x_1, x_2, ..., x_n, c, b))
   );
 end;
end;
```

A.6 Properties in redefinitions

A.6.1 *Predicates.*

A.6.1.1 *Reflexivity*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1, y_2 be Θ_{n+1};
 redefine pred π(y_1, y_2);
 reflexivity
 proof
  thus for a being Θ_{n+1} holds π(a, a);
 end;
end;
```

A.6.1.2 *Irreflexivity*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1, y_2 be Θ_{n+1};
 redefine pred π(y_1, y_2);
 irreflexivity
 proof
  thus for a being Θ_{n+1} holds not π(a, a);
 end;
end;
```

A.6.1.3 *Symmetry*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1, y_2 be Θ_{n+1};
```

```
 redefine pred π(y1, y2);
 symmetry
 proof
  thus for a, b being Θn+1 st π(a, b) holds π(b, a);
 end;
end;
```

A.6.1.4 *Asymmetry*

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn, y1, y2 be Θn+1;
 redefine pred π(y1, y2);
 asymmetry
 proof
  thus for a, b being Θn+1 st π(a, b) holds not π(b, a);
 end;
end;
```

A.6.1.5 *Connectedness*

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn, y1, y2 be Θn+1;
 redefine pred π(y1, y2);
 connectedness
 proof
  thus for a, b being Θn+1 holds π(a, b) or π(b, a);
 end;
end;
```

A.6.2 *Functors.*

A.6.2.1 *Commutativity*

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn, y1, y2 be Θn+1;
 redefine func ⊗(y1, y2);
 commutativity
 proof
  thus for a, b being Θn+1 holds ⊗(a, b) = ⊗(b, a);
 end;
end;
```

A.7 Definitional expansions

A.7.1 *Predicates.*

A.7.1.1 *Non permissive*

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 pred π(x1, x2, ..., xn) means :ident:
  Φ(x1, x2, ..., xn);
end;
```

$\pi(x_1, x_2, \ldots, x_n)$
proof
 thus $\Phi(x_1, x_2, \ldots, x_n)$;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 pred $\pi(x_1, x_2, \ldots, x_n)$ means :*ident*:
  $\Phi_1(x_1, x_2, \ldots, x_n)$ if $\Gamma_1(x_1, x_2, \ldots, x_n)$,
  $\Phi_2(x_1, x_2, \ldots, x_n)$ if $\Gamma_2(x_1, x_2, \ldots, x_n)$,
  $\Phi_3(x_1, x_2, \ldots, x_n)$ if $\Gamma_3(x_1, x_2, \ldots, x_n)$
  otherwise $\Phi_n(x_1, x_2, \ldots, x_n)$;
 consistency;
end;

$\pi(x_1, x_2, \ldots, x_n)$
proof
 per cases;
 case $\Gamma_1(x_1, x_2, \ldots, x_n)$;
  thus $\Phi_1(x_1, x_2, \ldots, x_n)$;
 end;
 case $\Gamma_2(x_1, x_2, \ldots, x_n)$;
  thus $\Phi_2(x_1, x_2, \ldots, x_n)$;
 end;
 case $\Gamma_3(x_1, x_2, \ldots, x_n)$;
  thus $\Phi_3(x_1, x_2, \ldots, x_n)$;
 end;
 case not $\Gamma_1(x_1, x_2, \ldots, x_n)$ & not $\Gamma_2(x_1, x_2, \ldots, x_n)$ & not $\Gamma_3(x_1, x_2, \ldots, x_n)$;
  thus $\Phi_n(x_1, x_2, \ldots, x_n)$;
 end;
end;

A.7.1.2 *Permissive*

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 assume $\Psi(x_1, x_2, \ldots, x_n)$;
 pred $\pi(x_1, x_2, \ldots, x_n)$ means :*ident*:
  $\Phi(x_1, x_2, \ldots, x_n)$;
end;

$\pi(x_1, x_2, \ldots, x_n)$
proof
 thus $\Psi(x_1, x_2, \ldots, x_n)$;
 thus $\Phi(x_1, x_2, \ldots, x_n)$;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 assume $\Psi(x_1, x_2, \ldots, x_n)$;

```
 pred π(x1, x2, ..., xn) means :ident:
  Φ1(x1, x2, ..., xn) if Γ1(x1, x2, ..., xn),
  Φ2(x1, x2, ..., xn) if Γ2(x1, x2, ..., xn),
  Φ3(x1, x2, ..., xn) if Γ3(x1, x2, ..., xn)
  otherwise Φn(x1, x2, ..., xn);
 consistency;
end;

π(x1, x2, ..., xn)
proof
 thus Ψ(x1, x2, ..., xn);
 per cases;
 case Γ1(x1, x2, ..., xn);
  thus Φ1(x1, x2, ..., xn);
 end;
 case Γ2(x1, x2, ..., xn);
  thus Φ2(x1, x2, ..., xn);
 end;
 case Γ3(x1, x2, ..., xn);
  thus Φ3(x1, x2, ..., xn);
 end;
 case not Γ1(x1, x2, ..., xn) & not Γ2(x1, x2, ..., xn) & not Γ3(x1, x2, ..., xn);
  thus Φn(x1, x2, ..., xn);
 end;
end;
```

A.7.2 *Modes.*

A.7.2.1 *Non permissive*

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 mode μ of x1, x2, ..., xn -> Θ means :ident:
  Φ(x1, x2, ..., xn, it);
 existence;
end;

a is μ of x1, x2, ..., xn
proof
 thus Φ(x1, x2, ..., xn, a);
end;

definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 mode μ of x1, x2, ..., xn -> Θ means :ident:
  Φ1(x1, x2, ..., xn, it) if Γ1(x1, x2, ..., xn),
  Φ2(x1, x2, ..., xn, it) if Γ2(x1, x2, ..., xn),
  Φ3(x1, x2, ..., xn, it) if Γ3(x1, x2, ..., xn)
  otherwise Φn(x1, x2, ..., xn, it);
 existence;
```

```
 consistency;
end;
```

```
a is μ of x1, x2, ..., xn
proof
 per cases;
 case Γ1(x1, x2, ..., xn);
  thus Φ1(x1, x2, ..., xn, a);
 end;
 case Γ2(x1, x2, ..., xn);
  thus Φ2(x1, x2, ..., xn, a);
 end;
 case Γ3(x1, x2, ..., xn);
  thus Φ3(x1, x2, ..., xn, a);
 end;
 case not Γ1(x1, x2, ..., xn) & not Γ2(x1, x2, ..., xn) & not Γ3(x1, x2, ..., xn);
  thus Φn(x1, x2, ..., xn, a);
 end;
end;
```

A.7.2.2 *Permissive*

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 assume Ψ(x1, x2, ..., xn);
 mode μ of x1, x2, ..., xn -> Θ means :ident:
  Φ(x1, x2, ..., xn, it);
 existence;
end;
```

```
a is μ of x1, x2, ..., xn
proof
 thus Ψ(x1, x2, ..., xn);
 thus Φ(x1, x2, ..., xn, a);
end;
```

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 assume Ψ(x1, x2, ..., xn);
 mode μ of x1, x2, ..., xn -> Θ means :ident:
  Φ1(x1, x2, ..., xn, it) if Γ1(x1, x2, ..., xn),
  Φ2(x1, x2, ..., xn, it) if Γ2(x1, x2, ..., xn),
  Φ3(x1, x2, ..., xn, it) if Γ3(x1, x2, ..., xn)
  otherwise Φn(x1, x2, ..., xn, it);
 existence;
 consistency;
end;
```

```
a is μ of x1, x2, ..., xn
proof
```

```
 thus Ψ(x_1, x_2, ..., x_n);
 per cases;
 case Γ_1(x_1, x_2, ..., x_n);
  thus Φ_1(x_1, x_2, ..., x_n, a);
 end;
 case Γ_2(x_1, x_2, ..., x_n);
  thus Φ_2(x_1, x_2, ..., x_n, a);
 end;
 case Γ_3(x_1, x_2, ..., x_n);
  thus Φ_3(x_1, x_2, ..., x_n, a);
 end;
 case not Γ_1(x_1, x_2, ..., x_n) & not Γ_2(x_1, x_2, ..., x_n) & not Γ_3(x_1, x_2, ..., x_n);
  thus Φ_n(x_1, x_2, ..., x_n, a);
 end;
end;
```

A.7.3 *Attributes.*

A.7.3.1 *Non permissive*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 attr x_n is α means :ident:
  Φ(x_1, x_2, ..., x_n);
end;
```

```
x_n is α
proof
 thus Φ(x_1, x_2, ..., x_n);
end;
```

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 attr x_n is α means :ident:
  Φ_1(x_1, x_2, ..., x_n) if Γ_1(x_1, x_2, ..., x_n),
  Φ_2(x_1, x_2, ..., x_n) if Γ_2(x_1, x_2, ..., x_n),
  Φ_3(x_1, x_2, ..., x_n) if Γ_3(x_1, x_2, ..., x_n)
  otherwise Φ_n(x_1, x_2, ..., x_n);
 consistency;
end;
```

```
x_n is α
proof
 per cases;
 case Γ_1(x_1, x_2, ..., x_n);
  thus Φ_1(x_1, x_2, ..., x_n);
 end;
 case Γ_2(x_1, x_2, ..., x_n);
  thus Φ_2(x_1, x_2, ..., x_n);
 end;
```

```
 case Γ3(x1, x2, ..., xn);
  thus Φ3(x1, x2, ..., xn);
 end;
 case not Γ1(x1, x2, ..., xn) & not Γ2(x1, x2, ..., xn) & not Γ3(x1, x2, ..., xn);
  thus Φn(x1, x2, ..., xn);
 end;
end;
```

A.7.3.2 *Permissive*

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 assume Ψ(x1, x2, ..., xn);
 attr xn is α means :ident:
  Φ(x1, x2, ..., xn);
end;

xn is α
proof
 thus Ψ(x1, x2, ..., xn);
 thus Φ(x1, x2, ..., xn);
end;

definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 assume Ψ(x1, x2, ..., xn);
 attr xn is α means :ident:
  Φ1(x1, x2, ..., xn) if Γ1(x1, x2, ..., xn),
  Φ2(x1, x2, ..., xn) if Γ2(x1, x2, ..., xn),
  Φ3(x1, x2, ..., xn) if Γ3(x1, x2, ..., xn)
  otherwise Φn(x1, x2, ..., xn);
 consistency;
end;

xn is α
proof
 thus Ψ(x1, x2, ..., xn);
 per cases;
 case Γ1(x1, x2, ..., xn);
  thus Φ1(x1, x2, ..., xn);
 end;
 case Γ2(x1, x2, ..., xn);
  thus Φ2(x1, x2, ..., xn);
 end;
 case Γ3(x1, x2, ..., xn);
  thus Φ3(x1, x2, ..., xn);
 end;
 case not Γ1(x1, x2, ..., xn) & not Γ2(x1, x2, ..., xn) & not Γ3(x1, x2, ..., xn);
  thus Φn(x1, x2, ..., xn);
 end;
```

```
end;
```

A.8 Identify

A.8.1 *without* ***when*** *statement*

```
registration
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 identify τ1(x1,x2,...,xn) with τ2(x1,x2,...,xn);
 compatibility
 proof
  thus τ1(x1,x2,...,xn) = τ2(x1,x2,...,xn);
 end;
end;
```

A.8.2 *with* ***when*** *statement*

```
registration
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 let y1 be Ξ1, y2 be Ξ2, ..., yn be Ξn;
 identify τ1(x1,x2,...,xn) with τ2(y1,y2,...,yn)
  when x1 = y1, x2 = y2, ..., xn = yn;
 compatibility
 proof
  thus x1 = y1 & x2 = y2 & ... & xn = yn
   implies τ1(x1,x2,...,xn) = τ2(y1,y2,...,yn);
 end;
end;
```

B. THE SYNTAX OF THE MIZAR LANGUAGE

```
Mizar Language Syntax

Last modified: November 24, 2010

System terms:
      File-Name,
      Identifier,
      Numeral,
      Symbol.



********        Article

Article = Environment-Declaration Text-Proper .



********        Environment
```

```
Environment-Declaration = ‘‘environ’’ { Directive } .

Directive = Vocabulary-Directive | Library-Directive |
   Requirement-Directive .

Vocabulary-Directive = ‘‘vocabularies’’ Vocabulary-Name
   { ‘‘,’’ Vocabulary-Name } ‘‘;’’ .

Vocabulary-Name = File-Name .

Library-Directive =
 ( ‘‘notations’’ |
   ‘‘constructors’’ |
   ‘‘registrations’’ |
   ‘‘definitions’’ |
   ‘‘theorems’’ |
   ‘‘schemes’’ ) Article-Name { ‘‘,’’ Article-Name } ‘‘;’’ .

Article-Name = File-Name .

Requirement-Directive =
   ‘‘requirements’’ Requirement { ‘‘,’’ Requirement } ‘‘;’’ .

Requirement = File-Name .



********        Text Proper

Text-Proper = Section { Section } .

Section = ‘‘begin’’ { Text-Item } .

Text-Item =
   Reservation | Definitional-Item | Registration-Item |
   Notation-Item | Theorem | Scheme-Item | Auxiliary-Item |
   Canceled-Theorem .

Reservation = ‘‘reserve’’ Reservation-Segment
   { ‘‘,’’ Reservation-Segment } ‘‘;’’ .

Reservation-Segment = Reserved-Identifiers ‘‘for’’ Type-Expression .

Reserved-Identifiers = Identifier { ‘‘,’’ Identifier } .

Definitional-Item = Definitional-Block ‘‘;’’ .
```

```
Registration-Item = Registration-Block ‘‘;’’ .

Notation-Item = Notation-Block ‘‘;’’ .

Definitional-Block = ‘‘definition’’ { Definition-Item | Definition }
   [ Redefinition-Block ] ‘‘end’’ .

Redefinition-Block = ‘‘redefine’’ { Definition-Item | Definition } .

Registration-Block = ‘‘registration’’
   { Loci-Declaration |  Cluster-Registration |
     Identify-Registration |  Canceled-Registration }
   ‘‘end’’ .

Notation-Block =
   ‘‘notation’’ { Loci-Declaration | Notation-Declaration }
   ‘‘end’’ .

Definition-Item =
   Loci-Declaration | Permissive-Assumption | Auxiliary-Item .

Notation-Declaration = Attribute-Synonym | Attribute-Antonym |
   Functor-Synonym | Mode-Synonym | Predicate-Synonym |
   Predicate-Antonym .

Loci-Declaration =
   ‘‘let’’ Qualified-Variables [ ‘‘such’’ Conditions ] ‘‘;’’ .

Permissive-Assumption = Assumption .

Definition = Structure-Definition | Mode-Definition |
   Functor-Definition | Predicate-Definition |
   Attribute-Definition | Canceled-Definition .

Structure-Definition = ‘‘struct’’ [ ‘‘(‘‘ Ancestors ‘‘)’’ ]
   Structure-Symbol [ ‘‘over’’ Loci ]
   ‘‘(#’’ Fields ‘‘#)’’ ‘‘;’’ .

Ancestors =
   Structure-Type-Expression { ‘‘,’’ Structure-Type-Expression } .

Structure-Symbol = Symbol .

Loci = Locus { ‘‘,’’ Locus } .

Fields = Field-Segment { ‘‘,’’ Field-Segment } .
```

```
Locus = Variable-Identifier .

Variable-Identifier = Identifier .

Field-Segment =
   Selector-Symbol { ‘‘,’’ Selector-Symbol } Specification .

Selector-Symbol = Symbol .

Specification = ‘‘->’’ Type-Expression .

Mode-Definition = ‘‘mode’’ Mode-Pattern
   ( [ Specification ] [ ‘‘means’’ Definiens ] ‘‘;’’
       Correctness-Conditions |
   ‘‘is’’ Type-Expression ‘‘;’’ ) .

Mode-Pattern = Mode-Symbol [ ‘‘of’’ Loci ] .

Mode-Symbol = Symbol | ‘‘set’’ .

Mode-Synonym = ‘‘synonym’’ Mode-Pattern ‘‘for’’ Mode-Pattern ‘‘;’’ .

Definiens = Simple-Definiens | Conditional-Definiens .

Simple-Definiens =
   [ ‘‘:’’ Label-Identifier ‘‘:’’ ] ( Sentence | Term-Expression ) .

Label-Identifier = Identifier .

Conditional-Definiens = [ ‘‘:’’ Label-Identifier ‘‘:’’ ]
   Partial-Definiens-List
   [ ‘‘otherwise’’ ( Sentence | Term-Expression ) ] .

Partial-Definiens-List =
    Partial-Definiens { ‘‘,’’ Partial-Definiens } .

Partial-Definiens = ( Sentence | Term-Expression ) ‘‘if’’ Sentence .

Functor-Definition = ‘‘func’’ Functor-Pattern [ Specification ]
   [ ( ‘‘means’’ | ‘‘equals’’ ) Definiens ] ‘‘;’’
   Correctness-Conditions { Functor-Property } .

Functor-Pattern = [ Functor-Loci ] Functor-Symbol [ Functor-Loci ] |
   Left-Functor-Bracket Loci Right-Functor-Bracket .

Functor-Property = ( ‘‘commutativity’’ | ‘‘idempotence’’ |
```

```
   ‘‘involutiveness’’ | ‘‘projectivity’’ )
   Justification ‘‘;’’ .

Functor-Synonym =
   ‘‘synonym’’ Functor-Pattern ‘‘for’’ Functor-Pattern ‘‘;’’ .

Functor-Loci = Locus | ‘‘(‘‘ Loci ‘‘)’’ .

Functor-Symbol = Symbol .

Left-Functor-Bracket = Symbol | ‘‘{‘‘ | ‘‘[‘‘ .

Right-Functor-Bracket = Symbol | ‘‘}’’ | ‘‘]’’ .

Predicate-Definition =
   ‘‘pred’’ Predicate-Pattern [ ‘‘means’’ Definiens ] ‘‘;’’
   Correctness-Conditions { Predicate-Property } .

Predicate-Pattern = [ Loci ] Predicate-Symbol [ Loci ] .

Predicate-Property = ( ‘‘symmetry’’ | ‘‘asymmetry’’ |
   ‘‘connectedness’’ | ‘‘reflexivity’’ | ‘‘irreflexivity’’ )
   Justification ‘‘;’’ .

Predicate-Synonym =
   ‘‘synonym’’ Predicate-Pattern ‘‘for’’ Predicate-Pattern ‘‘;’’ .

Predicate-Antonym =
   ‘‘antonym’’ Predicate-Pattern ‘‘for’’ Predicate-Pattern ‘‘;’’ .

Predicate-Symbol = Symbol | ‘‘=’’ .

Attribute-Definition =
   ‘‘attr’’ Attribute-Pattern ‘‘means’’ Definiens ‘‘;’’
   Correctness-Conditions .

Attribute-Pattern =
   Locus ‘‘is’’ [ Attribute-Loci ] Attribute-Symbol .

Attribute-Synonym =
   ‘‘synonym’’ Attribute-Pattern ‘‘for’’ Attribute-Pattern ‘‘;’’ .

Attribute-Antonym =
   ‘‘antonym’’ Attribute-Pattern ‘‘for’’ Attribute-Pattern ‘‘;’’ .

Attribute-Symbol = Symbol .
```

```
Attribute-Loci = Loci | ‘‘(‘‘ Loci ‘‘)’’ .

Canceled-Definition = ‘‘canceled’’ [ Numeral ] ‘‘;’’ .

Canceled-Registration = ‘‘canceled’’ [ Numeral ] ‘‘;’’ .

Cluster-Registration = Existential-Registration |
   Conditional-Registration |
   Functorial-Registration .

Existential-Registration =
   ‘‘cluster’’ Adjective-Cluster Type-Expression ‘‘;’’
   Correctness-Conditions .

Adjective-Cluster = { Adjective } .

Adjective = [ ‘‘non’’ ] [ Adjective-Arguments ] Attribute-Symbol .

Conditional-Registration = ‘‘cluster’’ Adjective-Cluster ‘‘->’’
   Adjective-Cluster Type-Expression ‘‘;’’
   Correctness-Conditions .

Functorial-Registration = ‘‘cluster’’ Term-Expression ‘‘->’’
   Adjective-Cluster [ Type-Expression ] ‘‘;’’
   Correctness-Conditions .

Identify-Registration =
   ‘‘identify’’ Functor-Pattern ‘‘with’’ Functor-Pattern
   [ ‘‘when’’ Locus ‘‘=’’ Locus { ‘‘,’’ Locus ‘‘=’’ Locus } ] ‘‘;’’
   Correctness-Conditions .

Correctness-Conditions = { Correctness-Condition }
   [ ‘‘correctness’’ Justification ‘‘;’’ ] .

Correctness-Condition = ( ‘‘existence’’ | ‘‘uniqueness’’ |
   ‘‘coherence’’ | ‘‘compatibility’’ | ‘‘consistency’’ )
   Justification ‘‘;’’ .

Theorem = ‘‘theorem’’ Compact-Statement .

Scheme-Item = Scheme-Block ‘‘;’’ .

Scheme-Block = ‘‘scheme’’ Scheme-Identifier
   ‘‘{‘‘ Scheme-Parameters ‘‘}’’ ‘‘:’’ Scheme-Conclusion
   [ ‘‘provided’’ Scheme-Premise { ‘‘and’’ Scheme-Premise } ]
   Reasoning ‘‘end’’ .
```

```
Scheme-Identifier = Identifier .

Scheme-Parameters = Scheme-Segment { ‘‘,’’ Scheme-Segment } .

Scheme-Conclusion = Sentence .

Scheme-Premise = Proposition .

Scheme-Segment = Predicate-Segment | Functor-Segment .

Predicate-Segment =
   Predicate-Identifier { ‘‘,’’ Predicate-Identifier }
   ‘‘[‘‘ [ Type-Expression-List ] ‘‘]’’ .

Predicate-Identifier = Identifier .

Functor-Segment = Functor-Identifier { ‘‘,’’ Functor-Identifier }
   ‘‘(‘‘ [ Type-Expression-List ] ‘‘)’’ Specification .

Functor-Identifier = Identifier .

Auxiliary-Item = Statement | Private-Definition .

Canceled-Theorem = ‘‘canceled’’ [ Numeral ] ‘‘;’’ .

Private-Definition = Constant-Definition |
   Private-Functor-Definition |
   Private-Predicate-Definition .

Constant-Definition = ‘‘set’’ Equating-List ‘‘;’’ .

Equating-List = Equating { ‘‘,’’ Equating } .

Equating = Variable-Identifier ‘‘=’’ Term-Expression .

Private-Functor-Definition =
   ‘‘deffunc’’ Private-Functor-Pattern ‘‘=’’ Term-Expression .

Private-Predicate-Definition =
   ‘‘defpred’’ Private-Predicate-Pattern ‘‘means’’ Sentence .

Private-Functor-Pattern =
   Functor-Identifier ‘‘(‘‘ [ Type-Expression-List ] ‘‘)’’ .

Private-Predicate-Pattern =
   Predicate-Identifier ‘‘[‘‘ [ Type-Expression-List ] ‘‘]’’ .
```

```
Reasoning = { Reasoning-Item }
   [ ‘‘per’’ ‘‘cases’’ Simple-Justification ‘‘;’’
     ( Case-List | Suppose-List ) ] .

Case-List = Case { Case } .

Case = ‘‘case’’ ( Proposition | Conditions ) ‘‘;’’
   Reasoning ‘‘end’’ ‘‘;’’ .

Suppose-List = Suppose { Suppose } .

Suppose = ‘‘suppose’’ ( Proposition | Conditions ) ‘‘;’’
   Reasoning ‘‘end’’ ‘‘;’’ .

Reasoning-Item = Auxiliary-Item | Skeleton-Item .

Skeleton-Item = Generalization | Assumption |
   Conclusion | Exemplification .

Generalization = ‘‘let’’ Qualified-Variables
   [ ‘‘such’’ Conditions ] ‘‘;’’ .

Assumption = Single-Assumption | Collective-Assumption |
   Existential-Assumption .

Single-Assumption = ‘‘assume’’ Proposition ‘‘;’’ .

Collective-Assumption = ‘‘assume’’ Conditions ‘‘;’’ .

Existential-Assumption = ‘‘given’’ Qualified-Variables
   ‘‘such’’ Conditions ‘‘;’’ .

Conclusion = ( ‘‘thus’’ | ‘‘hence’’ ) Compact-Statement |
   Diffuse-Conclusion .

Diffuse-Conclusion = ‘‘thus’’ Diffuse-Statement |
   ‘‘hereby’’ Reasoning ‘‘end’’ ‘‘;’’ .

Exemplification = ‘‘take’’ Example { ‘‘,’’ Example } ‘‘;’’ .

Example = Term-Expression |
   Variable-Identifier ‘‘=’’ Term-Expression .

Statement = [ ‘‘then’’ ] Linkable-Statement | Diffuse-Statement .

Linkable-Statement = Compact-Statement | Choice-Statement |
   Type-Changing-Statement | Iterative-Equality .
```

```
Compact-Statement = Proposition Justification ‘‘;’’ .

Choice-Statement = ‘‘consider’’ Qualified-Variables
   ‘‘such’’ Conditions Simple-Justification ‘‘;’’ .

Type-Changing-Statement = ‘‘reconsider’’ Type-Change-List
   ‘‘as’’ Type-Expression Simple-Justification ‘‘;’’ .

Type-Change-List = ( Equating | Variable-Identifier )
   { ‘‘,’’ ( Equating | Variable-Identifier ) } .

Iterative-Equality = [ Label-Identifier ‘‘:’’ ]
   Term-Expression ‘‘=’’ Term-Expression Simple-Justification
   ‘‘.=’’ Term-Expression Simple-Justification
 { ‘‘.=’’ Term-Expression Simple-Justification } ‘‘;’’ .

Diffuse-Statement = [ Label-Identifier ‘‘:’’ ]
   ‘‘now’’ Reasoning ‘‘end’’ ‘‘;’’ .

Justification = Simple-Justification | Proof .

Simple-Justification = Straightforward-Justification |
   Scheme-Justification .

Proof = ( ‘‘proof’’ | ‘‘@proof’’ ) Reasoning ‘‘end’’ .

Straightforward-Justification = [ ‘‘by’’ References ] .

Scheme-Justification =
   ‘‘from’’ Scheme-Reference [ ‘‘(‘‘ References ‘‘)’’ ] .

References = Reference { ‘‘,’’ Reference } .

Reference = Local-Reference | Library-Reference .

Scheme-Reference = Local-Scheme-Reference |
   Library-Scheme-Reference .

Local-Reference = Label-Identifier .

Local-Scheme-Reference = Scheme-Identifier .

Library-Reference = Article-Name ‘‘:’’
   ( Theorem-Number | ‘‘def’’ Definition-Number )
   { ‘‘,’’ ( Theorem-Number | ‘‘def’’ Definition-Number ) } .
```

```
Library-Scheme-Reference =
   Article-Name ‘‘:’’ ‘‘sch’’ Scheme-Number .

Theorem-Number = Numeral .

Definition-Number = Numeral .

Scheme-Number = Numeral .

Conditions = ‘‘that’’ Proposition { ‘‘and’’ Proposition } .

Proposition = [ Label-Identifier ‘‘:’’ ] Sentence .

Sentence = Formula-Expression .



********        Expressions

Formula-Expression = ‘‘(‘‘ Formula-Expression ‘‘)’’ |
   Atomic-Formula-Expression |
   Quantified-Formula-Expression |
   Formula-Expression ‘‘&’’ Formula-Expression |
   Formula-Expression ‘‘or’’ Formula-Expression |
   Formula-Expression ‘‘implies’’ Formula-Expression |
   Formula-Expression ‘‘iff’’ Formula-Expression |
   ‘‘not’’ Formula-Expression |
   ‘‘contradiction’’ |
   ‘‘thesis’’ .

Atomic-Formula-Expression =
   [ Term-Expression-List ] Predicate-Symbol
      [ Term-Expression-List ] |
   Predicate-Identifier ‘‘[‘‘ [ Term-Expression-List ] ‘‘]’’ |
   Term-Expression ‘‘is’’ Adjective { Adjective } |
   Term-Expression ‘‘is’’ Type-Expression .

Quantified-Formula-Expression =
   ‘‘for’’ Qualified-Variables [ ‘‘st’’ Formula-Expression ]
      ( ‘‘holds’’ Formula-Expression |
                  Quantified-Formula-Expression ) |
   ‘‘ex’’ Qualified-Variables ‘‘st’’ Formula-Expression .

Qualified-Variables = Implicitly-Qualified-Variables |
   Explicitly-Qualified-Variables |
   Explicitly-Qualified-Variables ‘‘,’’
      Implicitly-Qualified-Variables .
```

```
Implicitly-Qualified-Variables = Variables .

Explicitly-Qualified-Variables =
   Qualified-Segment { ``,'' Qualified-Segment } .

Qualified-Segment = Variables Qualification .

Variables = Variable-Identifier { ``,'' Variable-Identifier } .

Qualification = ( ``being'' | ``be'' ) Type-Expression .

Type-Expression = ``(`` Type-Expression ``)'' |
   Adjective-Cluster Type-Expression |
   Radix-Type .

Structure-Type-Expression = ``(`` Structure-Type-Expression ``)'' |
   Adjective-Cluster Structure-Symbol
      [ ``over'' Term-Expression-List ] .

Radix-Type = Mode-Symbol [ ``of'' Term-Expression-List ] |
   Structure-Symbol [ ``over'' Term-Expression-List ] .

Type-Expression-List = Type-Expression { ``,'' Type-Expression } .

Term-Expression = ``(`` Term-Expression ``)'' |
   [ Arguments ] Functor-Symbol [ Arguments ] |
   Left-Functor-Bracket Term-Expression-List Right-Functor-Bracket |
   Functor-Identifier ``(`` [ Term-Expression-List ] ``)'' |
   Structure-Symbol ``(#'' Term-Expression-List ``#)'' |
   Variable-Identifier |
   ``{`` Term-Expression [ Postqualification ] ``:''
         Sentence ``}'' |
   Numeral |
   Term-Expression ``qua'' Type-Expression |
   ``the'' Selector-Symbol ``of'' Term-Expression |
   ``the'' Selector-Symbol |
   ``the'' Type-Expression |
   Private-Definition-Parameter |
   ``it'' .

Arguments = Term-Expression | ``(`` Term-Expression-List ``)'' .

Adjective-Arguments = Term-Expression-List |
   ``(`` Term-Expression-List ``)'' .

Term-Expression-List = Term-Expression { ``,'' Term-Expression } .
```

```
Postqualification = ‘‘where’’ Postqualifying-Segment
   { ‘‘,’’ Postqualifying-Segment } .

Postqualifying-Segment = Postqualified-Variable
   { ‘‘,’’ Postqualified-Variable } ‘‘is’’ Type-Expression .

Postqualified-Variable = Identifier .

Private-Definition-Parameter = ‘‘$1’’ | ‘‘$2’’ | ‘‘$3’’ | ‘‘$4’’ |
   ‘‘$5’’ | ‘‘$6’’ | ‘‘$7’’ | ‘‘$8’’ | ‘‘$9’’ | ‘‘$10’’ .
```

ACKNOWLEDGMENT

The authors wish to express their sincere appreciation to the outstanding work of Andrzej Trybulec, the inventor of Mizar and the leader of the Mizar project.

References

[1] G. Bancerek and P. Rudnicki. Information retrieval in MML. In *MKM'03: Proceedings of the Second International Conference on Mathematical Knowledge Management*, pp. 119–132, 2003.

[2] P. Corbineau. A declarative language for the Coq proof assistant. In *Types for Proofs and Programs*, LNCS 4941, pp. 69–84, 2008.

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

[4] A. Grabowski and A. Naumowicz. Computer Reconstruction of the Body of Mathematics *Studies in Logic, Grammar and Rhetoric*, 2009.

[5] J. Harrison. A Mizar Mode for HOL. In *TPHOLs'96: Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, pp. 203–220, 1996.

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

[7] A. Korniłowicz. How to define terms in Mizar effectively. In [4].

[8] R. Matuszewski and P. Rudnicki. Mizar: the first 30 years. *Mechanized Mathematics and Its Applications*, 4(1), pp. 3–24, 2005.

[9] R. Matuszewski and A. Zalewska. From Insight to Proof. Festschrift in Honour of Andrzej Trybulec. *Studies in Logic, Grammar and Rhetoric*, 2007.

[10] Mizar home page: `http://mizar.org`.

[11] A. Naumowicz. Enhanced processing of adjectives in Mizar. In [4].

[12] A. Naumowicz. Teaching How to Write a Proof. In *Formed 2008: Formal Methods in Computer Science Education*, pp. 91–100, 2008.

[13] A. Naumowicz and C. Byliński. Improving Mizar texts with properties and requirements. In A. Asperti, editor, *MKM-2004*, LNCS 3119, pp. 290–301, 2004.

[14] A. Naumowicz and A. Korniłowicz. A Brief Overview of Mizar. In *S. Berghofer et al. (Eds.), TPHOLs 2009*, LNCS 5674, Springer-Verlag Berlin Heidelberg, 2009.

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

[16] QED Manifesto: `http://www.rbjones.com/rbjpub/logic/qedres00.htm`.

[17] Ch. Schwarzweller Mizar attributes. A technique to encode mathematical knowledge. In [9].

[18] D. Syme. Three tactic theorem proving. In *TPHOLs'99: Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, pp. 203–220, 1999.

[19] A. Trybulec. Some Features of the Mizar Language. In Proceedings of ESPRIT Workshop, Torino 1993.

[20] A. Trybulec. Tarski Grothendieck set theory. *Formalized Mathematics*, 1(1), pp. 9–11, 1990.

[21] J. Urban. XML-izing Mizar: Making Semantic Processing and Presentation of MML Easy. In *Mathematical Knowledge Management: MKM 2005*, LNCS 3863, pp. 346–360, 2006.

[22] M. Wenzel and F. Wiedijk. A comparison of Mizar and Isar. *Journal of Automated Reasoning*, 29(3-4), pp. 389–411, 2002.

[23] F. Wiedijk. Formal Proof Sketches. In *Types for Proofs and Programs: TYPES 2003*, LNCS 3085, pp. 378–393, 2004.

[24] F. Wiedijk. Mizar Light for HOL Light. In *Theorem Proving in Higher Order Logics: TPHOLs 2001*, LNCS 2152, pp. 378–393, 2001.

Contents