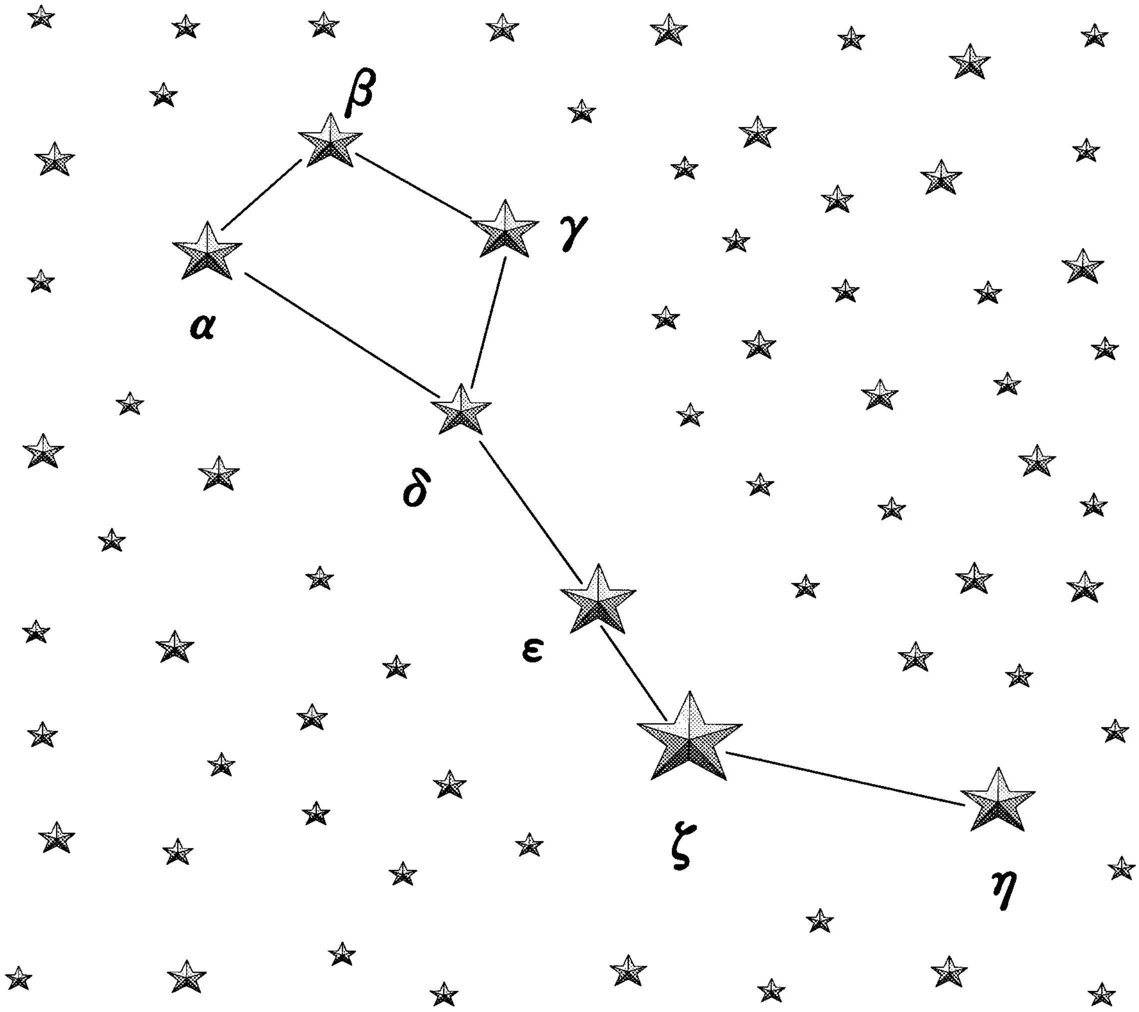
# *Mizar Lecture Notes*

(4th Edition：Mizar Version 6.1.12)

Yatsuka Nakamura, Toshihiko Watanabe
Yasushi Tanaka, Pauline Kawamoto



Shinshu University,
Dept. of Information Engineering, Kiso lab.

Mizar Lecture Notes

Published in March 16th, 1994
Revised Edition Published in July 29th, 1996.
4th Edition Published in Oct 1st, 2002.

Authors: Yatsuka Nakamura, Toshihiko Watanabe, Yasushi Tanaka, Pauline Kawamoto
Publisher: Shinshu University, Department of Information Engineering, Kiso Lab.

Published by Shinshu University, Department of Information Engineering, Kiso Lab.
4-17-1 Wakasato Nagano-shi, Nagano-ken 380-8533
Shinshu University, Department of Information Engineering
Nakamura Research Lab, Information Engineering Course
TEL 026 (269) 5495

©1994 Yatsuka Nakamura

# Table of Contents

# Preface

Mizar is one of the most famous proof checkers. It is historically very old and also thought to be the most accurate for genuine mathematical language. There seem to be several types of proof checkers in the world; however, those were made by graduate students for their graduation thesis and many of them were abandoned once the students graduated. Because of this situation, Mizar's library is increasing with the help of the Mizar group lead by Prof. Andrzej Trybulec as well as many other researchers and graduate students.

Since Prof. Trybulec is a mathematician, he knows about what kinds of proof checkers are needed. Therefore, because these were made under such experiences, their functions are thought to be substantial. This proof checker was non-existent in the old days; therefore, mathematics were thought to be very rigid. Many people thought that dependence in machines were not necessary. However, corresponding with the development of computers, this rigidity became questionable. As a result, a question rose as to how mathematics should be written. Every proof checker has different ideas. My creation, THEAX, is one of them; however, it was totally independent from Mizar even though it was created about the same time or a little later than Mizar. Therefore, the way of thinking is entirely different.

Here, we would like to explain Mizar's way of thinking, the actual environment of movement, and the story about what kind of thinking made formalized mathematics.

# Concerning the Revised Edition

Today's Mizar is beginning to be used increasingly in a large field of applications and it came to have a WWW homepage on the Internet. With support from the users and for the sake of its processing ability, Mizar has managed to renew its version frequently. We hope to help people who are learning Mizar by summarizing those improvements as well as we can.

# Concerning the 4th Edition

The content of this note is based upon Mizar version 6.1.12.

# Chapter 1

# A Summary of Mizar

Mizar is a general term of a project that formalized the use of computers by the Mizar society lead by Prof. Andrzej Trybulec of Warsaw University in Poland. The Mizar project is carried out by describing mathematical demonstrations by the Mizar language. Mizar was created to describe mathematics by using computers and checking them with a Mizar proof checker on a IBM-PC for Mizar library registration. The purpose of this project is to create a system for checking mathematical thesis. In Mizar, text that have descriptions of mathematical demonstrations are called article. When we write new articles, we can refer to articles that have been previously checked and registered in the Mizar library. When that article is registered in the Mizar library, other articles will be able to refer to it.

The articles are divided into an environ and main sections. In the environ section, we describe the necessary environmental set up for that article and in the main section, the demonstrations themselves are described.

The articles are described by the Mizar language. The Mizar language is based on the description method of general demonstrations; however, in Mizar, there is a specific method for writing. This will be explained later in more detail.

We will explain the Mizar language, Mizar proof checker, and the Mizar project.

# Chapter 2

# An Outline of the Mizar Language

The Mizar language is used for describing mathematical demonstrations by using computers. Because we normally follow a method of writing demonstrations when we write a Mizar article, we have to write with the Mizar language. From now, we abbreviate Mizar language to Mizar.

In Mizar, we use general ASCII code. Letters that are usually used are numbers, English alphabet, symbols. Also, for mathematical symbols, there are ones that use similar things or combine general ASCII characters .

Also, in Mizar, we have to keep in mind that capital and lower-case letters of the English alphabet are distinguished. Therefore, when we write MS-DOS names of articles and file names into an article, we always have to use capital letters. In this case, we write within 8 capital English letters, numbers, and, _, '.

As we use such words as 'therefore', 'theorem', 'definition', and 'end of demonstration' when writing mathematical demonstrations; in Mizar, we also use corresponding words to describe. These words become reserved words. As a Mizar system, the logic of `&`, `or`, `not`, concept of =, and predicate logic of All, Exist are prepared from the beginning. Moreover, we press on with our demonstrations by phrasing definitions and theorems of functor, predicate, and others that are registered in the Mizar library; however, their basic is written in `HIDDEN.ABS`. The materials that are written here is accepted as axioms without demonstrations. We structured the Mizar library by phrasing those and leading different theorems. When users write new articles, they describe by phrasing logic that are already prepared in the system from the beginning or the numerous theorems that are created based upon things such as `TARSKI`.

When users define functor function and others, they first register those into the vocabulary file and then define them in an article. In this case, we cannot define a functor that has exactly the same name with a reserved word. The method for defining specifically is explained in chapter 6. Also, please look at the table of Mizar reserved words in the appendix.

We will now discuss the logical formula and predicate logic that are prepared in the Mizar system. We can use the basic logical formula such as `&`, `or`, `not`, `implies`, and `iff`. For example, the following can be written.

```
A & B
A = B & B = C impliesA = C
A & ( B or C ) iff A & B or A & C
```

`&` is used for the logical formula. For between the formulas, we use and. Also, `implies` and `iff` indicate 'then' and if only if 'equivalence'.

For the predicate logic, we can use all symbols and exist symbols like the following.

```
for x st f[x] = b holds g[x] = c;
```

For the writing method of the normal predicate logic, this indicates the following formula.

$$(\forall x)(f(x) = b \Rightarrow g(x) = c)$$

This means for an optional x, if $f(x) = b$, then $g(x) = c$. Therefore, it means that for all x, $g(x) = c$ that satisfies $f(x) = b$ will be formed.

Further, `f[x]`, `g[x]` are the predicates that include `x` and it does not mean that this kind of writing method is prepared in the system.

Next, if we write the following,

```
ex x st f[x] =b & g[x] =c ;
```

it means the formula below.

```
(∃x)(f (x) =b and g (x) =c )
```

It means that x that becomes *g (x) = c* (when *f(x) =b*) exists. Also, we can mix all and existent symbols.

```
for x ex y st f[x] = a & h [x] = i [y]
```

The above formula indicates the following:

$$(\forall x)(\forall y)(f(x) = a \,\&\, h(x) = i(y))$$

We are allowed to use parentheses as an option. Even when parentheses are not needed, we can use them for better understanding.

In addition, there are the MS-DOS version and the Linux version in the Mizar system. Furthermore, although it is essentially the Linux version, there is a Mizar system which can be used for freedom from a Web page.

# Chapter 3

# The Structure of the Mizar Article

## 3.1 The Whole Structure

Next, we will explain the structure of an article. An article consists of two parts, the environ and main sections. The environ section starts with `environ` and ends with a semicolon. The main section starts with `begin` and ends with a semicolon. However, we can also divide a main section into some parts by inserting several "`begin`"s. This is equivalent to dividing into a paragraph in a paper. If the name of a paragraph is written to the `.BIB` file which we explain later, when an article is automatically changed into the form of the general mathematics paper later, the name of a paragraph will be written in a `begin` position.

```
environ
            The environ section
                                      ;
begin
           The main section 1
                                      ;
begin
           The main section 2
------
begin
           The main section n
                                      ;
```

## 3.2 The Environ Section

We will first explain the environ section. In the environ section, the necessary environmental set up for an article is conducted. Specifically, we write definitions, theorems, and vocabularies that the article phrases, and declare file names of the article that have them.

The environ section is constructed of the following items.

```
environ
    vocabulary     ・・・;
    notations      ・・・;
    constructors  ・・・;
    definitions    ・・・;
    theorems       ・・・;
    schemes        ・・・;
    requirements  ・・・;
```

We will explain each item in the environ section.

### 3.2.1 Vocabulary Section

In the vocabulary section, we write the names of the vocabulary files (`.VOC` file) that are registered and used in articles with capital letters. We omit the extension unit of a file name, and when writing many vocabulary files, we write by using commas (for vocabularies that are registered in `HIDDEN.VOC`, we can use them without writing them in the vocabulary section). At the end of sentences, we write semicolons. Also, we write the same way with places other than the vocabulary section.

The vocabulary file that has registered vocabularies that are used in an article registered in the Mizar library is summed up in a special file.

To look for the vocabulary file that has registered vocabularies that we want to use, we type the following at the command prompt of MS-DOS, or the prompt of Linux.

```
FINDVOC [vocabulary]
```

Also, when we want to indicate vocabularies that are registered in the vocabulary file, we type the following.

```
LISTVOC [vocabulary]
```

### 3.2.2 Notation Section and Constructors Section

In case of making a new theory, we sometimes want to use an old concept. For example, suppose that we want to use the concept of `Function` or `Function-like`. At this time, since these terms are registered into a vocabulary file called `FUNC.VOC`, we add `FUNC` to the vocabulary section. Moreover, since these terms are defined in `FUNCT_1` (`MIZ` or `ABS` file), we add `FUNCT-1` to the notation section. And in many cases, we add `FUNCT_1` also to the constructors section.

Example;

```
environ
   vocabulary FUNC;
   notation FUNCT_1;
   constructors FUNCT_1;
 begin
 reserve X for Function;
 theorem X is Function-like;
```

About the difference between the notation section and the constructors section, there was only one item called the signature section in the version of old MIZAR and both were unified. From there, relations such as the hierarchic structure between concepts and the definition of each concept were expanded by the memory of a computer. However, since the former had many overlapping parts, it was separated for saving memory. The former was inserted into the constructors section. Therefore, when there was a file with more inclusive structure, we could omit the file of smaller structure from the constructors section, and it came to be able to perform saving of memory. For example, even if as follows instead of the previous example, an error does not come out.

```
environ
 vocabulary FUNC;
 notation FUNCT_1;
 constructors TOPREAL1;

begin
reserve X for Function;
theorem X is Function-like;
```

Therefore, when TOPREAL1 is contained in the constructors section beforehand, we don't have to insert FUNCT_1 there.

At first, we can insert a file name into both the notation section and the constructors section in order, after completion, if a more basic file is deleted from the constructor section, we can simplify the environ section more, and we can also save memory.

In the notation section, when we want to use symbols written in the files, we write the file names of articles that have those symbols defined. In the definition section (part of definition) of an article, those symbols' meanings are added (things like what kind of argument they have and how many, or what kind of modes they themselves become).

For example, ∩(intersection) is defined in both of the articles BOOLE and

PRE_TOPC (the latter is redefined). However, the method of using that symbol is different.

```
< article BOOLE >                  < article PRE_TOPC >


definition                         definition
  let X,Y;                           let TP, P, Q;
  func X /\ Y -> set means           redefine
  x c= it iff x in X & x in Y;       func P /\ Q -> Subset of TP;
end;                               end;
```

/\ (same as ∩) in BOOLE means the set itself, and it has a set on the both sides as an argument. On the other hand, it in PRE_TOPC makes itself a Subset of TP(topological space), and it has Subsets of TP on both sides as an argument.

From now on, when we want to write articles using /\, if we want to use that as a set, we write BOOLE in the notation section, and if we want it to be a Subset of TP, we have to write PRE_TOPC.

### 3.2.3 Requirements Section

We put only a file called ARYTM on the requirements section at present. ARYTM is a file for using arithmetic operation comparatively freely.

For example, although the following formula becomes truth without giving a reason, this is the favor of ARYTM (however, a reason is required for 132-24 = 108).

```
132 + 24 = 156;
```

For example, the following is a no error. Multiplication and inequality are also judged automatically. Let's keep in mind that ARYTM is contained also in notation.

```
environ
  vocabulary REAL_1;
  notation ARYTM,REAL_1;
  constructors REAL_1;
  requirements ARYTM;
begin
theorem 10+4=14 & 10+5=15 & 10+6=16 & 10+7=17;
theorem 10+10=20 & 1000+1000=2000 & 10000+10000=20000;
theorem 44 * 2=88;
theorem 2 * 1=2 & 3 * 2=6 & 4 * 5=20;
theorem 45≦50;
theorem 45＜50;
```

## 3.2.4 Definitions Section

In the definitions section, when we want definition extension and expansion possible, we write file names of articles that have their definitions. For instance, when demonstrating `X c= Y`, we just need to demonstrate `∀a(a in X implies a in Y)`； however, this definition is defined in the article `TARSKI` as below.

```
pred X c= Y means x in X implies x in Y;
```

Therefore, if we write `TARSKI` in the definitions section, when we demonstrate `X c= Y`, it means that we demonstrated `X c= Y` if we can say that `a in X implies a in Y`.

Example,

```
theorem X c=X \/ Y
```

```
proof let x be set;
      assume  x in X;
      hence y in X \/ Y by BOOLE:def 2;
end:
```

### 3.2.5 Theorems Section

In the theorems section, we write the file names of articles with theoremsand definitions for citing articles.

### 3.2.6 Schemes Section

In the schemes section, we write the file names of articles that have phrasing schemes (demonstration form) written.

## 3.3 The Main Section

### 3.3.1 Sentences

Next, we will explain the main section. The main section basically consists of sentences and their line up. Sentences have the following structure.

```
 L1: A=B      by … ;
 L2: B=C      by … ;
 L3: A=C        by L1,L2;
Label Formula      Citation Section
```

We will attach only the necessary sentences (sentences that will be cited later) to labels. Also, we call `by ...` a citation section, and write labels of sentences that are necessary to lead that formula. In the citation section, `by ...`, we can mix labels in an article and cited labels from other articles. For example, labels in that article are untouched and when citing from other articles, we write a name of an article : a number as shown below.

```
······ by L1,L2,TARSKI:5;
```

Also, in case of citing a definition, we write a name of an article : def a number.

Moreover, for proof ~ end, instead of the citation section, we can add a demonstration with some sentences like the following.

```
L3:  A=C
proof
  ...
  ...
 hence thesis;
end;
```

In this case, all of these are considered as one sentence. At this point, end in proof ~ end needs thus or hence just before it. It means that this can be said about the first formula that has a demonstration attached currently by this sentence's formula. Concerning the difference between thus and hence: hence takes the form of then + thus.

Even though then has appeared here for the first time, it is used when a formula stated before is cited. To explain with an example, let us assume the following.

```
L1:A=B by ··· ;
L2:B=C by ··· ;
```

At this point, instead of writing the following, because L2 formula is a formula stated before, we will take L2 away from the citation section of L3 sentence.

```
L3:A=C by L1,L2;
```

By using then, we can write the following.

```
then L3:A=C by L1;
```

The reason why we use `then` is because we want to minimize the usage of labels. Normally with demonstrations, we often cite formulas stated before. Also, when citing formulas from the one stated before, we will need many labels if we attach labels and state `by` label. However, by using `then` here, we will not need to use labels when citing formulas from the one stated before. Therefore, we can reduce the number of labels. Thus, when we want to cite formulas from the one stated before, even though it is not wrong to write by using labels, it is better to use `then`.

In addition, when checking with a checker (explained in Chapter 8.1.1) for improving an article used for registering in the Mizar library, everything will return an error if we write with the method of using the above labels even though the demonstrations are right. As long as that error exists, it will not be possible to register in the Mizar library. Therefore, it should not be written. Where `then` can be used, it should be used instead of labels. Because of this, `hence` should be used for citing formulas just before the end of `proof ~ end`.

### 3.3.2 Theorem

Generally, the main section is structured centering theorems and definitions.

When writing a definition with an article, it should be written in the following way.

```
theorem   a formula from the theorem
 proof
```

```
   ...
   ...
  end;
```

The difference between the method of writing the above and the below is that by writing a `theorem`, that formula is pulled out as an abstract file.

```
Formula
 proof
   ...
   ...
  end;
```

An abstract file is a file that has a `theorem`, `definition`, and others that are pulled out of an article (`.ABS` file). When we register an article in the Mizar library, we change an article into an abstract file. The reason for this is because articles that contain everything such as demonstrations (`.MIZ` file) become too big, and it is hard to find anything for phrasing. Therefore, when we write formulas that we desire to be cited, we put in a `theorem`.

When we write definitions with an article, we use a `definition`. Also, when some new words or functions are to be defined, this `definition` is used to define them. We will explain definitions later in more detail.

Further, a theorem or definition cannot be written in `proof` ~ `end`. We can only use them in the top step of a nest. In short, we cannot write as follows.

```
theorem ...
 proof
  theorem ...
   proof
   ...
  ...
  end;
end;
```

As explained, a Mizar article is largely divided into the environ section and the main section. The main section is usually structured with some definitions and theorems.

Also, in a Mizar article, after :: is a comment.

# Chapter 4

# Method for Demonstration

## 4.1 Basic Method for Demonstration

We will explain the demonstration method. For a demonstration method, as we have explained before, we will first write a formula that needs to be demonstrated and then we will write the demonstration in between `proof` and `end`.

```
A formula that is needs to be demonstrated
   proof
    ...
    ...
    thus (hence) ...;
   end;
```

The end of a demonstration will always finish with `thus...` or `hence....`

Let us demonstrate a proposition "`p implies q`".

```
p implies q
 proof
  assume p;       p is assumed
   ...
   ...
  thus q;         q could be indicated
 end;
```

First, we will initially write a formula "p implies q" that needs to be demonstrated, and write proof , end;. We do not need a semicolon after proof; however, there has to be one after end. If it is p, we need to demonstrate that it is q; however, p is also one logical formula. We will first write assume p and then p is assumed, and if we can indicate q, the demonstration is finished with thus q.

Next, if we want to demonstrate "b=c implies a=b" after a=c is assumed, the following can be stated. With assume a=c, we will place label A:.

```
A:a=c;
 b=c implies a=b
 proof
  assume B:b=c
  hence thesis by A;
 end;
```

With assume ..., we hypothesize .... In short, we indicate that it is a hypothesis section. Also, we should indicate that for hence ..., the last formula of this demonstration could be derived from a formula just before hence and the reasoning section of a hence formula. A thesis means that a formula should be indicated.

Let's explain thus and hence a little in more detail.

Although thus and hence are used in proof, the formula which should be proved becomes easy every time hence and thus appear and a proved part is removed from the formula connected by &. For example, the following is the proof of three formulas connected by and.

```
theorem 3-1=2 & 5-2=3 & 6-3=3
proof 2+1=3;
 hence 3-1=2 by REAL_2:17;
 3+2=5;
 hence 5-2=3 by REAL_2:17;
 3+3=6;
 hence thesis by REAL_2:17;
end;
```

Since `3-1=2` are proved by the first `hence`, we become the formula which `5-2=3 & 6-3=3` should prove. Since `5-2=3` are proved by the next `hence`, `6-3=3` become the formula which should finally be proved. It means that all the remaining parts (`6-3=3`) are first proved by `hence thesis`.

In addition, `hence` combines `then` and `thus`, as mentioned above.

```
A1: A=B;
A2: B=C;
thus A=C by A1,A2;
```

For example, we can write the above as the following.

```
A1: A=B;
B=C;
hence A=C by A1;
```

## 4.2 Irrational Method

When demonstrating the same thing, there is also this method of doing it. When `p implies q` needs to be demonstrated, we still assume the condition with `assume p` in the `proof`. Then we will deny the conclusion by stating `assume not q`.

```
p implies q
 proof
  assume p;
  assume not q;
  ......
  thus contradiction;
 end:
```

And then we will lead from this and state `thus contradiction`. It means that a irrational method was used. If we assume that it is neither `p` nor `q`, it means that an inconsistency occurred.

## 4.3 Dividing the Demonstrations

There are also many different formations for demonstrations. These are called skeletons of demonstrations.

For instance, let us assume that we want to demonstrate `p & q & r`. At this point, the following demonstrations can also be done. (This paragraph becomes the repeat of 4.1.)

```
p & q & r
 proof
 ......
 thus p;
 ......
 thus q;
 ......
 thus r;
```

```
  ......
end;
```

First, let us assume that `p` could be demonstrated and it becomes `thus p`. Also, continuing with the demonstration, it becomes `thus q`, when we demonstrate even further, it becomes `thus r`. In this case, the Mizar system looks for `thus .....` For this, three `thus` appear so that of a proposition `p & q & r` that should be demonstrated first, we will recognize the fact that `p` could be demonstrated. Also, we will recognize that the second one could be demonstrated with `thus q`. Finally, we will be aware that everything could be demonstrated with `thus r`. The end can also be stated as thus theses. In this case, system and `p,q` are already demonstrated and only `r` is left; therefore, we will recognize that `r` is the `thesis`. In this way, we will be aware that `thus thesis` is `thus r`. It is fine to write either one of them.

In addition, when demonstrating `p & q & r`, we can either demonstrate `p` first then `q & r` next, or `p & q` first and `r` later. We can demonstrate by dividing freely.

## 4.4 Demonstration of the Equivalence

Concerning the demonstrating method for `p iff q (`*p* `<=>` *q*`)`, `p implies q` is demonstrated first, then `q implies p` is demonstrated. The system automatically interprets `p iff q` as `p implies q & q implies p`. Therefore, if we state that `p implies q`, the first half is considered to be demonstrated. The rest is the other half's demonstration. As a result, with `assume q`, it becomes `thus p` and it is judged that `p iff q` is said.

```
p iff q
 proof
  assume p;
   ...
  thus q; p => q could be said
  assume q;
   ...
  thus p; q => p can be said and p <=> q
 end:
```

## 4.5 Method for Using Now

For a skeleton demonstration, `now` is sometimes used to demonstrate `not p`. `Now` is used as follows:

```
now assume p;
 .....
 .....
 thus contradiction;
end;
```

With `now ~ end`, `now` is considered as one sentence (logical formula). This `now ~ end` is the same with `not p`. Or, with `assume p`, it becomes a logical formula `p` that implies a contradiction.

## 4.5.1 Caution About End

As a caution about end, in Mizar, end is used in many ways such as `proof ~ end` and `now ~ end`. These have been nesting. When there are many `end`s, we get confused on which `end` is which. Therefore, because `end` is needed when `now` is written, we should write `end` as soon as writing `now`. The rest is just filling this in with an editor. In the same way, we should write `end` as soon as a `proof` is written. We will not make any mistakes if we just remember that `end` will not be written automatically.

## 4.6 All Symbols

We will now discuss about a predicate proposition. Let us demonstrate `for x holds p[x] & q[x]`. Further, `p[x]` means a predicate that includes `x` and it does not mean that brackets are permitted.

```
for x holds P[x] & Q[x]
 proof
  let x;
   ......
   ......
  thus P[x]:
   ......
  thus Q[x]; (thesis)
 end:
```

The word let exists as an opposite of `for` from all symbols. `P` is demonstrated by thus `P[x]` with leaving let `x` as it is. Further, we will demonstrate `Q` with `Q[x]`.

If it does not get confused, this letter can be changed. For example, the letter can be changed to `y`.

### 4.6.1 When There Are Two Variables

When there are two variables like the following we write two variables.

```
for x,y st P[x,y] holds Q[x,y]
 proof
  let x,y;
  assume P[x,y];
   ......
```

```
    ......
   thus Q[x,y]; (thesis)
  end;
```

Directly, it means that we demonstrated $(\forall x)(\forall y)(P(x,y) \Rightarrow Q(x,y))$. "`assume .....` " becomes one logical formula.

### 4.6.2 Method for Writing With such that

In addition, there is another way of writing as follow:

```
for x,y st P[x,y] holds Q[x,y]
 proof
  let xs,y such that L:P[x,y];
   ......
   ......
   ......
  thus Q[x,y];
 end;
```

Further, labels are usually entered after `such that`; however, in this example (this case `L`) they are not necessary. The first line is `st` and the third line is `such that`.

### 4.6.3 Division of Sentences

There is a skeleton such as this:

```
for x,y st P[x,y] & Q[x.y] holds R[x,y]
 proof
  let x,y such that L1:P[x,y] and L2:Q[x.y];
   ......
```

```
      ......
     thus R[x,y];
    end:
```

We can also rewrite this as the following. We need to pay careful attention to the difference of & and and.

The third line

```
    let x,y such that L1:P[x,y] and L2:Q[x,y];
```

is

```
     let x,y such that L:P[x,y] & Q[x,y];
```

We can also write like the above. Methods for writing the front and back are basically the same. In the method for writing the front, and is used for dividing into two sentences to build other labels. This means that one sentence A & B was divided into sentences A and B. On the other hand, the method for writing the back is persistently one sentence and it is connected by a logical calculation unit. As stated before, we can divide one sentence into two by using and, or, we can connect two sentences into one by using &.

## 4.6.4 Dividing Sentences With Assume

For the difference in writing a method after such that, a similar thing occurs in the case of assume. For example, adding & S[x,y] to an assume example, we will assume that the following occurred.

```
    for x,y st P[x,y] & S[x,y] holds Q[x,y]
```

```
proof
 let x,y;
 assume P[x,y] & S[x,y];
  ......
  ......
 thus Q[x,y];
end;
```

At this point,

```
 assume P[x,y] & S[x,y];
```

can be written as below.

```
assume that L1:P[x,y] and L2:S[x,y];
```

Only with the way of writing the above formula, we can write labels on both `P[x,y]` and `S[x,y]`. In this example, they are `L1` and `L2` labels.

### 4.6.5 Caution Following that

When using `that`, we cannot use `then` after `that`. It is because we think there may be more than two sentences after `such that`. When there are more than two sentences, the sentence just before them is not clear.

```
assume that A and B:
then C;
```

Therefore, the above method of writing is not possible. We have to use labels and write in the following way.

```
assume that L1:A and L2:B;
C by L1,L2;
```

Or, when only `B` is used, we can only write as below:

```
C by L2;
```

Also, when there is only one citation, `then` cannot be used. We have to write by using `L2` instead of `then`. Therefore, when `that` is stated, labels have to be there or this sentence can not be used. In short, this will not be able to be cited later. Thus, we have to think that the labels will always follow `that`.

When `that` is not used, `then` can be used. Therefore, the first way of writing, `assume P[x,y] & Q[x,y]`, in this case, the following can be used.

```
assume P[x,y] & Q[x,y];
then ......
```

Thus, there is only one sentence without `that`. For `assume`, `then` is used.

## 4.7 Existing Symbols

There are more different kinds of skeletons. In the case of demonstrating logical predicate formulas with existing symbols `ex x st P[x]`, we use `take`.

By using `take`, we write the following:

```
ex x st P[x]
 proof
  ------
  L:P[a];
  take a:
  thus thesis by L;
 end;
```

When demonstrating the existence of `x` that satisfies `P[x]`, let us assume that `P[a]` is demonstrated. In this case, it means that for `take a`, `x` is exists and if we take a, it is demonstrated by `L`.

### 4.7.1 Method for Using Consider

Consider is an opposite word for `take`.

```
(ex x st P[x]) implies for y st Q[y] holds R[y]
 proof
  assume ex x st P[x];
  then consider a such that L:P[a]
  ......
  ......
  let y such that Q[y];
  ......
  ......
  thus R[y];
 end;
```

First, we will assume. For `assume ex x st P[x];`, we use `consider` and state `then consider a such that L:P[a]`. This case also needs labels because `such that` exists. We will leave it like this and to state `y st` next, we will state `let y such that Q[y]` to state `thus Q[y]`.

`Consider` means to give some sort of proper nouns to an existing `x` that satisfies `P[x]`. Therefore, we intend to have an existing `x` as a topic for discussion. Thus, it does not have to be `x`, it can be `a`.

As an existing `a`, it can be used as a proper noun here. In other words, it is used as if it is assumed to exist. Therefore, `consider` is an opposite of `take`. `Take` is used for replacing something as a proper noun or bring in existing symbols.

## 4.7.2 Method for Using Given

There is another skeleton that is slightly different.

```
(ex x st P[x]) implies for y st Q[y] holds R[y]
 proof
  given x such that P[x];
  ......
  ......
  let y
  assume Q[y];
  ......
  ......
  thus R[y];
 end;
```

Here, given in the second sentence: "given x such that P[x];", works as both assume and consider.

Please remember as follows. It is much easier.

```
given = assume + consider
```

## 4.8 Dividing into Cases

Next, we will discuss about demonstrating by dividing into cases. Let us assume that we want to demonstrate a proposition "A and B" and it can be done if we divide these into cases. For instance, we want to demonstrate by dividing into cases such as when x=1, x=2, and when x takes some other value. This means that with x=1,2, it becomes a peculiar point. At this point, it takes the structure of the following demonstrations.

```
L:( x=1 or x=2 or not ( x=1 & x=2 ))
 now per cases by L;
  case LA:x=1;
   ......
  thus A and B;

  case LB:x=2;
   ......
  thus A and B;

  case LC:not (x=1 & x=2);
   ......
  thus A and B;
```

```
        end;
```

We will demonstrate by supposing each `case` in "`now per cases ~ end`". We will demonstrate from "`case x=1`" to the beginning of "`thus A and B`" when `x=1`. In the same way, we will also demonstrate when `x=2` and other cases. The sentences from now to end are considered as the whole one "`A and B`". For example, if we have then after "`now ~ end`", "`then`" receives the whole "`now ~ end`". Even though it is divided into each case with "`case`", the conclusion of each case is handed over to the end. Therefore, when all of this is cited, the fact that it was divided with "`case`" is forgotten and the fact that only the conclusions are the same is checked. Thus, it means that this is one formula.

In addition, since a formula like L is logically clear, there may not be "`by L`" of "`now per cases by L`".

### 4.8.1 Premise for Dividing into Cases

We have some warnings for you here. This case is used as a premise when dividing a formula `x=1 or x=2 or not (x=1 & x=2)` ---(1) into cases. It means that this division of cases covers all cases. When dividing into cases in this way, we have to state that we are stating this about all cases.

In addition, like the upper example, when formation of formula (1) is clear, "`by`" behind "`now per cases`" is unnecessary.

### 4.8.2 Labels for Dividing Into Cases

Unquestionably, when demonstrating by dividing into cases, the labels of each case demonstration is only valid during the demonstration of that particular case. If we try to use that label for the demonstration of another rather than that specific demonstration case, an error saying that there is no label will appear.

### 4.8.3 Warning About Labels

Further, we will give a warning about labels. In Mizar, it is acceptable to use the same label; however, the ones just before are prioritized.

For example:

```
L1:......;
    ......
L1:......;
    ...... by L1; Here, L1 of just before is cited.
```

When we want to demonstrate something with this case, if we state by `L1`, the sentence of `L1` below it will be cited.

This is easily mistaken and when we put labels `L1`, `L2`, `L3`, ...... , we tend to forget until the labels were used. Also, let us assume that we put `L3` and when we state by `L3`, even if we think that it is `L3` and we can certainly infer from `L3`, there will be an error. This kind of mistake is made often at first. When this happens, there is one same label that is hiding if examined carefully. Sometimes, there is another one in a different place.

## 4.9 Method for Using @proof

When we write a long article in Mizar, we end up checking constantly with a Mizar checker. It takes time to check a long article; therefore, by skipping the check of definitions that have already been demonstrated, we can save some time when checking in Mizar.

To omit checking definitions and others, we replace `proof` with `@proof`. By doing this, the Mizar checker will not check that definition.

Of course, when articles are completed, all the `@proof` need to be removed.

## 4.10 Introduction of New Variable(Set, reconsider)

When we introduce a new variable, we write as follows.

```
set r1 = r + 2;
```

Therefor we can introduce the new variable `r1` as `(r + 2)`. The type will turn into real number type, if `r` is real number type. In the bloc after this sentence, we can use a variable `r1` freely (if it escapes from the bloc by `end`, `r1` will become null and void). When we want to introduce the new variable from which a type differs, we use `reconsider`. we write as follows.

```
reconsider n1 = r + 2 as Nat by A2;
```

In this case, it is needed to attach a reason that `r+2` becomes a natural number type. It is same in `Set` that the new variable is effective only in the bloc. We can also introduce two or more variables simultaneously like the following.

```
reconsider n1 = r + 2, n2 =r + 3 as Nat;
```

## 4.11 About Take

Although it becomes the repeat of Section 4.7, we explain in more detail about `take`.

```
      ex r bring Real st 1<r-1
                         -------(1)
```

When proving a formula (1), we write it as follows.

```
      theorem ex r being Real 1<r-1
       proof 1+1<3;then A1:1+1-1<3-1 by REAL_1:59;
        take 3;
        thus 1 < 3-1 by REAL_2:17,A1;
       end:
```

We have to keep in mind that the last formula is as follows instead of the form of (1).

```
        thus 1<3-1
```

It is because there is "`take 3`" in the middle of proof, therefore the part of existing symboles are removed, we should prove only a naked part.

If we explain it in a little more detail, it is as below.

To prove (∃x)(∃y)(f(x,y)),

```
proof  ------ ------  a formula which should be proved in this stage is
                                                      (∃x)(∃y)(f(x,y))
       take a ------  a formula which should be proved in this stage is
                                                          (∃y)(f(a,y))
       ------ ------
       take b ------  a formula which should be proved in this stage is
                                                              (f(a,b))
       ------ ------
       hence f(a,b);
end;
```

Therefore, we can do the following proof.

```
theorem ex r,s being Real st 1<r-1
  proof take 3;
  take 1;1<3;
  hence thesis by REAL_2:17;
end;
```

Further, if we prove the part about the above-mentioned f(a,b) even if we don't use take in many cases, finally the checker will look for a and b. For example, we can also do it as follows.

```
theorem ex r being Real st 1<r-1
  proof 1+1<3;then 1+1-1<3-1 by REAL_1:59; then
   1<3-1 by REAL_2:17;
   hence thesis;
  end;
```

## 4.12 Hereby

`Hereby` used in proof is the same as `thus + now`.

For example, at the following the proposition of `(for x st x in A holds x in B)` is first proved and it will be removed from the proposition group that should be proved.

```
theorem TT1: (for x st x in A holds x in B) & A=A \/ A
proof
 hereby let x; assume X in A;
 hence x in B;
 end;
 thus A=A \/ A by BOOLE:35;
end;
```

Thus, the following part is the same as `(for -----)` above.

```
now let x;
 assume X in A;
 ------
 hence x in B;
end;
```

"This is proved first" means that we can attach thus before now as follows.

```
thus now ---
----
----
end;
```

This become as follows.

```
hereby ----
-----
-----
end;
```

There are the following ways of using `hereby` as a application course (This does not need to prove and is an obvious theorem.).

```
theorem AA2: x=y iff not y<>x
proof
 hereby assume x=y;
  hence not y<>x;
 end:
 assume not y<>x;
 hence x=y;
end:
```

The proposition `(x=y iff not y<>x)` is the same as a proposition called `(x=y implies not y<>x)&(not y<>x implies x=y)`. The former `(-----)` part is proved and removed by `hereby`, and we are proving the latter `(-----)` part in the second half.

As the further application course, there is the following.

```
theorem BB2: A=B
 proof hereby let x be Any; assume A1: x in A;
  thus x in B by TT1.A1;
 end:
 let x;
 assume X in B;
 hence x in in A by TT1;
```

```
      end;
```

This is regarded as (A=B) being the same as (A c=B)&(B c=A), because the definition section of an environ section has BOOLE. Furthermore, because the definintion section has TARSKI, it is considered that this is the same as the following.

```
      (for x st x in A holds x in B) &
      (for x st x in B holds x in A)
```

If it becomes this form, it will turn out that the above-mentioned example can describe by hereby. Actually, When we describe that two sets are equallike this last example, hereby is often used.

# Chapter 5

# About Mode

## 5.1 Mode

Next, we will explain mode. In Mizar, there is a concept of mode. Mode is a type of Pascal (a kind of computer language) and there is set for a general mode. Other than this, there are `Nat`, `Real`, and others as representative modes. Users can also define a mode. Also, as shown in the following figure, mode has a structure level such as `Nat` in `Real` and `Real` in `set`. Functor and others are subordinated to mode. This is similar to a programming language. And it is considered that `Any` and `set` are the same.

<table>
<tr><td rowspan="12">Set</td><td rowspan="4">Real</td><td>Nat</td></tr>
<tr><td>Integer</td></tr>
<tr><td>……</td></tr>
<tr><td></td></tr>
<tr><td rowspan="3">Function</td><td></td></tr>
<tr><td></td></tr>
<tr><td></td></tr>
<tr><td rowspan="3">Top Space</td><td></td></tr>
<tr><td></td></tr>
<tr><td></td></tr>
<tr><td rowspan="2">……<br>……</td><td></td></tr>
<tr><td></td></tr>
</table>

Fig. 5.1 a level of mode

For example, let us think about the following example. Here, let us assume that + is only defined pertaining to Real.

```
reserve A,B for Real;
        C,D for Function;
L:C=A+B:
  ......
  D=A by...;
  then C=D+B by L;
```

Let us assume that C=A+B and D=A are demonstrated when A,B is Real and C is Function. At this point, if we substitute D=A to a formula C=A+B and make C=D+B by L, it will be an error. It is because D and B have different modes. A calculation unit + is only defined to Real and it is not defined to Function. Therefore, an error of unknown functor will appear.

This error can be taken care of by extending the mode of variable D. It will be fine if we can state that D is Function as well as Real.

## 5.2 Structure Level

As we mentioned in the begining, mode has the following structure level. Set is the widest mode, for example, there is Real below it and Nat below Real. In other words, Nat becomes set, Real becomes set, and Nat becomes Real. This way of thinking is similar to an Object Oriented Language. The same symbols are used from the top.

## 5.3 Mode change

For mode change, we use the word reconsider to conduct a mode change. (cf.4.10).

```
reconsider U=D as Nat by ...;
```

As shown above, by using `reconsider`, it is possible to change `D` of `Real`'s mode to `U` of `Nat`'s mode.

## 5.4 Declaration of Mode During a Demonstration

Many times, we declare mode variables during demonstrations. We will again enter a skeleton of a demonstration; for example, in the demonstration of `P ⊆ Q`:

```
P c= Q
 proof
  let x be set;
   assume x in P;
   ......
   thus x in Q;
 end;
```

At this point, it is stated as let `x` be set;.

be mode name

With the above, we can declare mode during a demonstration. Of course, we can also state `reserve x for set` before hand.

### 5.4.1 Warning About c=

Also, as a caution, we use `c=` for ⊆ in Mizar. However, when using `c=` , we need to leave a space in front and back.

### 5.4.2 Example of Actually Using a Definition

Furthermore, here, we demonstrate $(\forall x)(x \in P \Rightarrow x \in Q)$ and it does not mean that we demonstrated $P \subseteq Q$ . There is a mathematical distance between these two. However, the system understands the fact that both of them are equivalent by writing `TARSKI` in the `definitions` of the environ section as explained in the third chapter. It means that this is defined as a predicate in the abstract file of `TARSKI`.

`TARSKI` is structured from the definition which is the basis of the Mizar library; therefore, when writing articles, we write `TARSKI` at the `definitions` of the environ section.

# Chapter 6

# Definition

## 6.1 Definition

In this chapter, we will explain predicates, functor, and mode definitions.

## 6.2 Method for Writing the Vocabulary File

In Mizar, when we newly define functors, we need to register those functors into a vocabulary file. This is also true when defining predicates and mode. In the vocabulary file (.VOC file), we must always start writing from the beginning of the line and write what the name is as the first word. For example, if it is HIDDEN.VOC:

HIDDEN.VOC

```
Mset
MReal
MNat
K[:
L:]
O+ 32
O.
R<>
R<
R>
```

(parts extracted)

As shown above, it displays what it is in one word at the beginning of the line.

| Symbols | Things that are defined |
|---|---|
| M | Mode |
| O | Functor |
| R | Predicate |
| K | Left functor bracket |
| L | Right functor bracket |
| G | Structure |
| U | Selector |
| V | Attribute |

We will write names after one word in the beginning of the line without any space. In addition, for functor, we will leave a space and write in the order of precedence. The order of priority consists of integral numbers from 1 to 255 and bigger the number, higher the priority order. If we omit, it will be 64.

Thus, after registering to the vocabulary file, we will define with an article. Further, in case of newly registering a vocabulary, we need to undertake the following procedure to make sure that it is not already registered.

```
CHECKVOC [vocabulary]
```

## 6.3 Predicate Definition

We will first explain the predicate definition. For instance, considering `k` and `l` as natural numbers, let us assume that we want to define `k` as a factor of `l`. This is a predicate. Let us assume that we also want to define such that `k` is a factor of `l`. This has a form similar to a binomial calculation unit; however, it has two variables of `k` and `l`. The following shows how to write the definition for this case.

```
definition
let k,l be Nat;
pred k is_factor_of l
```

```
means
:FACTOR:
ex t being Nat st l=k*t
```

The Nat in let k,l be Nat indicates that the mode of k and l is a natural number. :FACTOR: is a label and this is used for citation later. The labels in the definition are put between the colons. When the abstract file is made by pulling it out of here, it will be changed to def with numbers. Labels of normal theorems will be changed to simple numbers. Also, * in k*t symbolizes multiplication.

As a result, a predicate of k is_factor_of l means ex t being Nat st l=k*t.

## 6.4 Functor Definition

We will explain the fuctor definition next. For example, let us think about a 2 variable fuctor. Let us assume that we want to make a new set Z from the set (set)X,Y. We will explain what to do at this point.

```
 X /\ Y → Z
set  set  set
```

```
definition
 let X , Y be set;
 func X /\ Y -> set
 means
 :N: x in it iff
    x in X & x in Y;
 existence ......;
 uniqueness
 proof
 end;
end;
```

This case also allows for it to be written in the operation form instead of the definition of a function like $f(X,Y)=Z$. Let us assume that we want to define such a symbol as $Z = X \cap Y$. To make a new set $Z$ from sets $X$ and $Y$ is truly this functor. With `let X, Y be set;`, we have sets $X$, $Y$ ready. In addition, `func X /\ Y -> set` indicates `X /\ Y` as a `set`. The next line after `means` is the meaning of that functor. We will then write the meaning of the functor by placing a label like before. In this meaning, `X /\ Y` is indicated as `it`.

In addition, in the demonstration section (contents of the meaning), we have to demonstrate the `existence` and `uniqueness`. In other words, we have to express its existence and uniqueness (the existence is only one).

For the demonstration for `uniqueness`, we only have to state the following.
If we take `A1` and `A2` as `it`:

```
    (x in A1 ⇔ x in X & x in Y)
and (x in A2 ⇔ x in X & x in Y)
 ⇒  A1 = A2
```

If we write this, it will be as below:

```
uniqueness
 proof
  let A1,A2 be set such that
   A1: x in A1 iff x in X & x in Y
   and
   A2: x in A2 iff x in X & x in Y;
   now let y;
```

```
      y in A1 iff y in X & y in Y by A1;
      hence y in A1 iff y in A2 by A2;
     end:
   hence A1=A2 by TARSKI:2;
```

In short, when we bring in the two sets A1 and A2 that satisfy this condition, by demonstrating the equality of these two sets, the uniqueness is in fact demonstrated.

In this demonstration of uniqueness, because it is such that in the third line, one can see that after it the two sentences label A1 and label A2 are connected with and. It is stated as now let y. Let us assume that this y is reserved as set. In addition, by label A1 we can state as y in A1 iff y in X & y in Y, and therefore, by label A2, y in A1 iff y in A2 can be stated. In addition, it means that enough necessary conditions for A1=A2 are demonstrated by TARSKI:2 stating that y in A1 and y in A2 are equally corresponding with the optional y; therefore, A1=A2 are demonstrated.

We also have to demonstrate the existence in the same way; however, we have to state that such an it exists. That is, let us demonstrate for the following.

```
  ex A being set st for x
  holds x in A1 iff x in X & x in Y
```

## 6.5 Method for Using Equals

If we use equals, such as $Q(x) = x^2$, we can define the function by substitution briefly. For example, if we descrive as follows, we can define the function Quard(x)=x*x.

```
   definition let x be Real;
    func Quard(x) equals :A1:
```

```
  x*x
  correctness;
 end;
```

## 6.6 Mode Definition

At last, we will explain the mode definition.

Let us assume that we will newly define a mode `nlt2_elements`.

```
definition
 mode nlt2_elements -> set means
 :DF2:
 ex a,b st a in it & b in it & a <> b;
  existence
  proof
   take B = NAT;
     thus ex a,b st a in B & b in B & a <> b
     proof
      take a= 1, b = 2;
      thus a in B & b in B & a <> b;
     end;
   end;
  end;
```

In the demonstration section of the mode definition, only the demonstration of `existence` will be needed. Also, when the mode is defined, its natural property can be defined by (attribute) or (cluster). For instance:

```
definition
```

```
attr nlt2_elements -> set means :DF2:
ex a,b st a in it & b in it & a <> b;
end;
definition
cluster nlt2_elements set;
existence
proof
 take B=NAT;
 ex a,b st a in B & b in B & a <> b
 proof
  take a=1, b=2;
  thus a in B & b in B & a <> b;
  end;
 hence thesis by DF2;
 end;
end;
```

Generally, attribute takes the following predicate form:

A is non-empty

Or, the adjective form.

non-empty set

In this case, non-empty is the same with non empty.
Also, in the definition blocks of attribute, we can define symbols that possess the same meaning (synonym) and the opposite meaning(antonym).

Example (this is not actual)

```
definition
```

```
 attr empty -&gt set means :A1:
  not ex x st x∈it;
  ...
 synonym (void);
 antonym (non-empty);
end;
```

When we want to use by attaching multiple attributes to one mode, that cluster needs to be defined. If this is not done, the following inconsistent combination will appear.

```
finite infinite set
empty non-empty set
```

As it has been stated before, new functor, mode, attribute, cluster (cf. §6.7), and others can be defined in Mizar; however, each demonstration needs to include the following elements.

| | Existence | Uniqueness |
|---|---|---|
| Mode | ○ | |
| Pred | | |
| Functor | ○ | ○ |
| Attr | | |
| Cluster | ○ | ○ |

## 6.7 Cluster Definition

We can make a type like new mode by giving some attribute to a certain mode. For example, suppose that there is mode called Real. Let us assume that the attribute of positive is defined to this mode. At this time, we can also make mode called "positive Real" (this is called cluster).

Although it seems we are enough to make a new mode called positive_Real, convenience of cluster is as follows:

```
x is positive_Real; then
x is positive;
```

We can not derive the above, but we can derive the following without giving reason.

```
x is positive Real; then
x is positive;
```

That is, the Mizar checker remembers firmly that cluster called "`positive Real`" is `positive Real`. If once new cluster is defined, we can use the cluster freely in other articles by writing the first filename such as clusters of the environ section. The example which defines cluster will be shown below. In the definition, let's keep in mind that it is necessary to prove only existence.
In addition, we will assume that the next is written in VOC file beforehand.

```
Vpositive
```

```
environ
 vocabulary TEST8;
 notation ARYTM,REAL_1;
 constructors ARYTM;
 theorems REAL_1,AXIOMS;
 requirements ARYTM;
begin
 definition let x be Real;
  attr x is positive means :A1: 0<x;
end;
definition
 cluster positive Real
 existence
 proof take 1; thus O<1; end;
end;
theorem for x,y begin positive Real
 holds x+y is positive Real
proof let x,y is positive Real;
 B1:0＜x by A1; then 0＜y by A1; then
 x<x+y by REAL_1:69; then
 0<x+y by B1, AXIOMS:22;
hence x+y is positive Real by A1;
end:
```

## 6.8 Introduction of structure

Let `(X,a)` be a binominal algebra. In such a case (X is a set and a is a function), we write only the following (without useing definition etc.).

```
struct Binalg(# base->set,op2->Function #);
```

For the new terminology `Binalg`, `base`, and `op2`, we write the following in a VOC file (be careful of adding the classification Symbol of `G`, `U`).

```
GBinalg
Ubase
Uop2
```

Then, the following theorem is formed without giving reason.

```
theorem for X being set, a being Function
holds (the base of Binalg(# X,a #))=X;
```

This corresponds to the proposition "the base of a binominal algebra `(X, a)` is X".

Take care of adding `the` to `base`.

# Chapter 7

# Practice Environment

Here, we explain the Practice Environment of installing and using the system of Mizar in our personal computer.

As a Practice Environment, although there are the MS-DOS version which we use at the MS-DOS prompt (command prompt) of Windows9x/2000/NT, and the Linux version which we use at Linux (kernel 2.0.x, 2.2.x, and 2.4.x) of the intel version, in any case, a certain editor is required. Since an extension letters is used in Mizar, please prepare an editor which can display an extension letters. In the MS-DOS version, we recommend an editor which is attached as a standard. And since we will refer to two or more library files, it is convenient to use an editor that we can open two or more editor windows.

## 7.1 Installation

Each file of MS-DOS version and the Linux version for Practice Environment installation of Mizar is about 14 M bytes archive file. Therefore, it is easy to get them using a browser or the ftp command from Homepage and the ftp server of Mizar. For example, if it is from Homepege, please get them from the following URL etc.

```
http://markun.cs.shinshu-u.ac.jp/kiso/projects2/proofchecker/mizar/index-j.html
```

Please refer to Appendix C about getting from an anonymous ftp server.

Mizar is used by installing the Mizar archive file to the hard disk.

### 7.1.1 MS-DOS Version

We can use Mizar of the MS-DOS version, installing to the hard disk drive of the AT compatible under the Windows 9x/2000/NT operating system. Besides, the free area of about 75 M bytes of hard disk drive is required. Here, we explain the stream which a Practice Environment builds from the archive file (for example, `mizar-6.1.11_3.33.722-win32.exe`) of Mizar to PC of Windows 98 whose hard disk drive is C drive.

We create a directory in a space of the hard disk drive (for example, `C:\work` etc.), and save the archive file of Mizar there.

We start an MS-DOS prompt from the start menu of Windows, and turn it English mode by the `us` command. And we change a current directory into the directory to which we saved the archive file of Mizar previously.

Since the archive file of Mizar is a self-extracting file, we input the file name of the acquired file as follows, and uncompresses it.

```
C:\work>mizar-6.1.11_3.33.722-win32
```

A batch file for installation called `INSTALL.BAT` is in the uncompressed files, we install Mizar by using this as follows.

```
C:\work>INSTALL .\ C:\MIZAR
```

Although `C:\MIZAR` is the directory where the system of Mizar is installed, we can also specify other drives and other directory names. The commands of Mizar are installed by this batch file, further, directories are created automatically and `MIZ` files, abstract files made until now are installed there.

After the installment is finished, `C:\MIZAR` is need to be added to the path. In addition, when it is specified except `C:\MIZAR` as the directory of the installation place of Mizar, we set the directory of the installation place of Mizar to the environment variable `MIZFILES` of a system.

To be more precise, when the installation place of Mizar is set to `C:\MIZAR`, we add the following one line to autoexec.bat, and reboot the PC.

```
PATH C:\MIZAR
```

By the way, when `PATH` already has been set to an `autoexec.bat` file like the following example,

```
PATH C:\WINDOWS\COMMAND
```

we use the semicolon and add the directory of Mizar to `PATH` as follows.

```
PATH C:\WINDOWS\COMMAND;C:\MIZAR
```

Moreover, when the installation place of Mizar is set to `D:\MIZAR` of the hard disk drive of `D` drive for example, we add the following two lines to `autoexec.bat` and reboot the PC.

```
PATH D:\MIZAR
set MIZFILES=D:\MIZAR
```

Besides, since these are explained in detail in the file called readme.txt made by uncompressing the archive file of Mizar, please refer to if needed.

### 7.1.2 Linux Version

We can use Mizar of the Linux version, installing to the hard disk drive of an AT compatible with the Pentium processor of Intel where Linux is installed (Mizar system runs on Linux with kernels 2.0.x, 2.2.x and 2.4.x). The free area of about 80 M bytes of hard disk drive is required. Here, we explain the stream which a Practice Environment builds from the archive file (for example, `mizar-6.1.11_3.33.722-linux.tar`) of Mizar to PC.

We create a directory (for example "`work`" on its own home directory) in a space of the hard disk drive, and save the archive file of Mizar there.

And we change a current directory into the directory to which we saved the archive file of Mizar previously.

Then, we extract the archive file of Mizar by the tar command. If # is a prompt, we enter the following.

```
# tar xvf mizar-6.1.12_3.35.723-linux.tar
```

A shell script file for installation called `install` is in the extracted files, we install Mizar by using this as follows.

```
# ./install
```

By default, the executable file of Mizar is installed in `/usr/local/bin`, the shared file of Mizar is tinstalled in `/usr/local/share/mizar` and the document file of Mizar is installed in `/usr/local/doc/mizar`. Therefore, it is necessary to do this work by the user who has a permission of writing to these directories when installing.

We can also install them to directories other than a default directory. In this case, we start the shell script for installation first, and specify a directory interactively on a screen during installation.

After the installment is finished, please check that the directories of the executable files of Mizar which have been installed are included in `PATH` of a system.

If it is a default case, we can check that /usr/local/bin is contained in `PATH` by entering as follows.

```
# echo $PATH
```

When not contained, after adding the following one line to `.bashrc` (in the case of `bash`),

```
export PATH=/usr/local/bin
```

we enter the following command and add `/usr/local/bin` to a command search path.

```
# source ~/.bashrc
```

Besides, when the executable file of Mizar is installed in directories other than default, it goes without saying that this directory is set to `PATH`.

Next, we set the directory of the installation place of shared file of Mizar to an environment variable `MIZFILES`.

To be more precise, when the installation place of Mizar is set to `/usr/local/share/mizar`,after adding the following one line to `.bashrc` (in the case of `bash`),

```
export MIZFILES=/usr/local/share/mizar
```

we enter the following command.

```
# source ~/.bashrc
```

Besides, since these are explained in detail in the file called `README` made by extracting the archive file of Mizar, please refer to if needed.

## 7.2 Preparation for Using Mizar

To use Mizar, we first need to make our own directory.

Here, we explaine an example according to a MS-DOS version. In `C:\`, a directory named `USER` needs to be created and our own directory is made under it. For instance, because my name is Nakamura, I need to make a directory named nakamura under the `USER` directory. Below it, three directories of `TEXT`, `DICT`, and `PREL` are created.

To explain each directory: `TEXT` directory is for placing Mizar articles (`.MIZ` file) that the users write. In addition, an intermediate file that outputs the Mizar system is also placed here. `DICT` is an abbreviation of DICTionary and the vocabulary file (`.VOC` file) that the user has written is placed. New words are written under this vocabulary file. `PREL` is an abbreviation of PRELiminaries, and this is used to store the necessary files to view articles that are not registered in the Mizar library.

```
C:\USER\NAKAMURA\TEXT
                \DICT
                \PREL
```

Usually, when using Mizar, we assume that a current directory is one's own directory under `C:\USER`. For example, when using the command of Mizar called `accom` which we explain by 7.4.1, we enter it as follows.

```
C:\USER\NAKAMURA\ACCOM TEXT\EXAMPLE1.MIZ
```

## 7.3 Mizar System

In case of the MS-DOS version, the Mizar system is made in Mizar C:\MIZAR; however, ABSTR is an important directory among these. ABSTR is an abbreviation of ABSTRact and this directory holds the abstract files that have been created up to this point in time. BOOLE.ABS and TARSKI.ABS are examples of these. When an article is written, files in here are read often.

There are the following as other directories.

In PREL, files that are used for viewing an article registered in the Mizar library are placed.

In MML, MIZ files (with detailed proof) of a library are installed in MML.

In DOC, readme file of this Mizar system, document files that are used for registering to the library, and other Mizar-related documents are installed.

Mizar's various commands are placed directly in C:\MIZAR.

Similarly, in case of the Linux version, as written also by the explanation of the installation procedure of 7.2, various commands of Mizar are installed in /usr/local/bin. In addition, the directories of ABSTR, PREL, or MML are made under /usr/local/share/mizar, DOC is made under /usr/local/doc/mizar, and files are installed there.

## 7.4 The Mizar Usage

### 7.4.1 Accommodate

For instance, let us assume that an article named EXAMPLE1.MIZ was created. This file is set on a directory called TEXT under one's own directories created by 7.2. First, we accommodate it (accommodate means to gather files and adjust them). In order to accommodate, at the command prompt, we set EXAMPLE1.MIZ as an argument and enter a command called ACCOM as follows.

```
ACCOM TEXT\EXAMPLE1.MIZ
```

An argument must be taken as a file name (here EXAMPLE1.MIZ) including a path. This accommodation reads the environ section in and gets ready for the article's proof by making some files of the referred article's theorems, definitions, and others that are needed by an article. Sometimes, errors may appear at this point when, for example, names of an abstract file are written incorrectly in the environ section.

Accommodation does not need to be carried out every time; however, when the environ section is changed, it needs to be accommodated again.

## 7.4.2 Mizar Checker

When the above is done, the Mizar checker is carried out next. To do this, the following form is needed.

```
VERIFIER TEXT\EXAMPLE1.MIZ
```

When this is done, a table will appear on the screen and checks an article in the order of Parser, Analyzer, and Checker and indicates the number of checked lines and errors. If only the checked number of rows is displayed after the check and there is no error, it means that it passed the checker of Mizar.

## 7.4.3 When There Is An Error

When there is an error, the error number is entered to an article in the form of comment by using a command called ERRFLAG. We use this command as follows.

```
ERRFLAG TEXT\EXAMPLE1.MIZ
```

Every command can omit an extension .MIZ.
To read an error message, another file (MIZAR.MSG) needs to be viewed with an editor.

### 7.4.4 Useful Command

There is a useful batch command named `MIZF.BAT` and when the following form takes place, it checks to see if there is any need for accommodation. When there is a need, it carries out the accommodation first and then the Mizar checker is executed.

```
MIZF TEXT\EXAMPLE1.MIZ
```

When there are more errors, it enters the error numbers in the form of a comment to an article. At the end of an article, risen error numbers and messages are entered into a comparison table (entered comments are erased at the time when `MIZF` command is executed next).

# Chapter 8

# Registering to the Library

We would like to explain the Mizar library registration process. Let us assume that we wrote a Mizar article and it passed the checker. However, we cannot register it to the Mizar library because it only passed the checker. Before registering, we have a few other things to do.

## 8.1 Improvement of an Article

### 8.1.1 Improvement of the Main Section

First, we need to improve an article. To do this, we need the following commands.

| | |
|---|---|
| `RELPREM` | Checks unnecessary citations. |
| `CHKLAB` | Checks unnecessary labels. |
| `INACC` | Checks unnecessary lines. |
| `TRIVDEMO` | Checks trivial demonstrations. |
| `RELINFER` | Checks unnecessary lines. |

Such commands are provided. All these commands are used in the following way.

```
RELPREM TEXT\EXAMPLE1.MIZ
```

If we execute the following after carrying out each command, the error number will be written into an article.

```
        ERRFLAG TEXT\EXAMPLE1.MIZ
```

RELPREM checks to see if irrelevant matters in reasoning are cited after "by". It also checks for the same thing for "then". If certain things are not needed for reasoning, an error will appear.

CHKLAB later on checks for the labels that are not used after labeling.

INACC later checks for the lines that are not used. If an error occurs from this checker and lines are erased, there may be excess citations and labels due to this; therefore, it needs to be checked again from the beginning.

TRIVDEMO checks to see if there are trivial proofs. Trivial matters need to be removed. When being as follows,

```
        A=B
        proof
        To:A=C by T1;
           C=D by T2;
        hence thosis by To;
        end;
```

it meens that we can simplify them like the following. T1 and T2 are the labels used between the above-mentioned proof and end (external label of proof --- end).

```
        A=B by T1,T2;
```

Explanation of RELINFER;

RELINFER checks to see if two lines are summarize to one. For example, When an error (*605) is displayed as follows,

```
        A=B + C by T1;then
```

```
D=E + (B + C) by T2;
            *605
```

it meens that we can manage with the following one line.

```
D=E + (B + C) by T1,T2;
```

When the error of the number `*604` is as follows,

```
S1: A=B + C by T1;
  ----------
  ----------
D=E+ (B + C) by S1,T2
            *604
```

it meens that we can write the following.

```
S1: A=B + C by T1;
  ----------
  ----------
D=E+(B + C) by T1.T2;
```

If the line of a label `S1` is not used for others, we can delete it, but if used, we cannot delete and cannot reduce a line. We can find out whether it is used or not, using `CHKLAB` once again, but when it is used, `*604` may be disregarded because of troublesome to restore it. Anyway, it is necessary to use a series of check programs repeatedly. As a result, the number of lines may be decreased no less than several 10 percent. Because derivation of a line to a line is powerful, it is natural for Mizar checker, but it may become an oversimplification which cannot be understood when man reads. Therefore, it is not need to necessarily turn the number of lines even into the minimum, and we may stop moderately.

### 8.1.2 Checking the Environ Section

In addition, we have to check the environ section. To do this, there are the following commands.

| | |
|---|---|
| IRRVOC | Checks unnecessary vocabulary. |
| IRRTHS | Checks unnecessary theorems. |

The method of using these is the same with the checking commands of the main section. For NOTATION and CONSTRUSTORS, there is no good way of checking. Therefore, use NOTATION and CONSTRUSTORS sparingly from the beginning and start with 0 and gradually include what is needed. However, if we accommodate after deleting an unnecessary file by IRRVOC, an error will appear in the NOTATION section, so we will be able to remove it.

## 8.2 Preparation for Registration

After checking for all the improvements in an article, MIZ2ABS is exectured next. When MIZ2ABS is executed, itis accommodated first. And then, the file for referring that article is made from one's article (.MIZ file), and only Theorem and Definition are picked out to make the abstract file (.ABS file). In addition, when MIZ2PREL is executed to make that article referable, it copies the necessary file into one's directory.

Command used after revision

| | |
|---|---|
| MIZ2ABS | Makes abstract files. |

`MIZ2PREL` Copies files so that the article can be seen.

In addition, when the article is cited, names of authors, their unit, their address, and references are written in normal sentences and made into a `.BIB` file. Please refer to a sample called `example.bib` in `DOC` directory.

The article is created with the process above and is sent to the Mizar Society on a floppy disk (for example). If this article is good enough, it will be accepted and an acceptance notice will be sent. When it is accepted, it will be automatically translated into English and will be published in the Formalized Mathematics magazine.

Besides, it us also sent to `mml@mizar.uwb.edu.pl` by using the attachment function of e-mail etc. Please refer to the method of submitting by e-mail in Appendix C.3.

Note:

1) When we want to quote a certain paper in SUMMERY, we write `/cite {A1}` in a sentence. We write the paper name into the space of REFERENCES independently, and give a code called `A1`.

2) Although PRESENTER in it means the major personalities of the referee of a pape, this space will be left blank.

# Chapter 9

# Useful tools

## 9.1 To find VOC files containing a terminology (findvoc)

In order to introduce new `terminology`, we make a new VOC file and register into it (cf. 6.2). In this case, when we want to know in which VOC file the already registered terminology is contained, we input as follows.

```
C:\>findvoc Real
```

By using this command, the list of VOC files with which all the terminology containing a character string called "Real" belongs is displayed.

If only this command is inputted as follows, how to use findvoc (help) will be displayed.

```
findvoc
```

For example, it is displaied how we should use the command like the following examples;

When we want to search for letter "`|`", we input as follows.

```
findvoc \b
```

And when we want to search for letter " <", we input as follows.

```
findvoc \l
```

## 9.2 To check the contents of the VOC file

Although all VOC files actually existed in the old version of MIZAR, they are unified and a user cannot see them individually now. So when we want to know the contents of a VOC file called `INDEX1.VOC`, it will be displayed by inputting as follows.

```
LISTVOC INDEX1
```

## 9.3 To serach for the ABS file which the terminology appears

### 9.3.1 Method for Using grep

We may want to know about where the terminology is used in the library. It is the case where there is a question (where to define) or a question (Where and where is the theorem about it). In such a case, although it is not contained in in the standard MIZAR system, it is convenient to use a program called egrep. For example, to list the ABS file in which terminology called `Metric` appears, we input as foolws.

```
C:\>cd /mizar/abstr
C:\>/mizar/egrep Metric *.* \| more
```

An `egrep.exe` program is free software (by Mr. M.Patnode) and we can get it from the following URL (of the Internet) .

```
http://www.eunet.bg/simtel.net/msdos/txtutl.html
```

It is convenient if we make it into the batch file including a change of a directory.

In addition, although it may be good to do like the above when searching for the general character string (alphanumeric character), the character string is to be searched for with a regular expression generally. Therefore, in order to search for the file and line which have the sum of three variables like a+b+c or x+y+z, we have to use it as follows.

```
egrep [a-z]\+[a-z]\+[a-z] *.*
```

Here, the following means one of the letters from Alphabet a to z, and \+ means the letter of +. And let us keep in mind that we have to write a sign like + or − after backslash \.

[a-z]

Detailed explanation of a regular expression is in large numbers on the Internet. For example, please refer to the next URL.

http://www.robelle.com/smugbook/regexpr.html

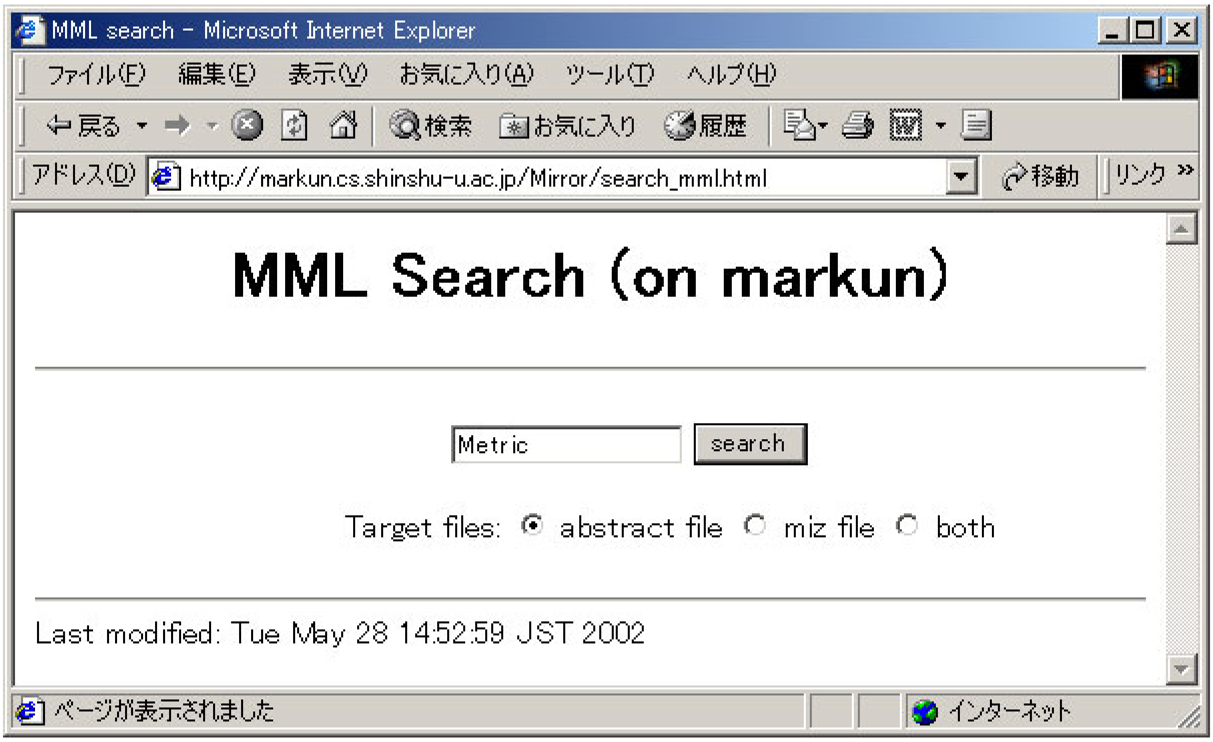
## 9.3.2 Method for serach on the Web

If we have the Internet access, we can search for it on the web.

Please open the next URL by using a browser. A search page as shown in Fig.9.1 is displayed on screen.

```
http://markun.cs.shinshu-u.ac.jp/Mirror/search_mml.html
```

On this page, first we enter a terminology to search (Metric in Fig.9.1). And we choose with a radio button where we search from (from abstract file, miz file or both)(abstract file in Fig.9.1). Then we can search by clicking the search button.



http://markun.cs.shinshu-u.ac.jp/Mirror/search_mml.html

MML Search (on markun)

Metric

Target files: ● abstract file ○ miz file ○ both

Last modified: Tue May 28 14:52:59 JST 2002

Fig.9.1 A search page on the Web

# Chapter 10

# About the version upgrade

Mizar versions are frequently updated. At this time, the increase of a library is natural and the check of its article under description is not affected, but we require care about a checker's version upgrade. Please watch the homepage of MIZAR on the Internet frequently and take care to get the newest.

Sometimes, the basic theorem which was being used before will be canceled and we may be puzzled. In this case, the information is written in the file in the directory of a new version. The name of this file is the following.

```
CANCELED.DOC
```

# Appendix A

# List of Reserved Words

```
and                antonym            attr
as                 assume             be
begin              being              by
canceled           case               cases
cluster            clusters           coherence
compatibility      consider           consistency
constructors       contradiction      correctness
def                deffunc            definition
definitions        defpred            end
environ            equals             ex
existence          for                from
func               given              hence
hereby             requirements       holds
if                 iff                implies
is                 it                 let
means              mode               not
notation           now                of
or                 otherwise          over
per                pred               proof
provided           qua                reconsider
redefine           reserve            scheme
schemes            set                st
struct             such               symmetry
synonym            take               that
the                then               theorem
theorems           thesis             thus
uniqueness         vocabulary         where
```

# Appendix B

# A Compilation of Mizar Library Theorems

Here, we have carried the theorems and definitions which were edited into the compact of article frequently quoted in the Mizar library.

Please refer to the original abstract file if needed. We can refer to it from the following URL.

`http://markun.cs.shinshu-u.ac.jp/Mirror/mizar.org/JFM/mmlident.html`

# B.1 TARSKI

```
Tarski Grothendieck Set Theory by Andrzej Trybulec

reserve x,y,z,u,N,M,X,Y,Z for set;

2 (for x holds x in X iff x in Y) implies X = Y;

def 1 func { y } means x in it iff x = y;
def 2 func { y, z } means x in it iff x = y or x = z;
def 3 pred X c= Y means x in X implies x in Y;
def 4 func union X means x in it iff ex Y st x in Y & Y in X;

7 x in X implies ex Y st Y in X & not ex x st x in X & x in Y;

scheme 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;

def 5 func [x,y] equals { { x,y }, { x } };
def 6 pred X,Y are_equipotent means
     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 holdsx = z iff y = u;
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 holdsY in Z) &
    (for X holds X c= M implies X,M are_equipotent or X in M);
```

## B.2 AXIOMS

```
Strong Arithmetic of Real Numbers by Andrzej Trybulec

reserve x,y,z for real number;
reserve i,k for Element of NAT;

13 x + (y + z) = (x + y) + z;
16 x * (y * z) = (x * y) * z;
18 x * (y + z) = x * y + x * z;
19 ex y st x + y = 0;
20 x <> 0 implies ex y st x * y = 1;
21 x <= y & y <= x implies x = y;
22 x <= y & y <= z implies x <= z;
24 x <= y implies x + z <= y + z;
25 x <= y & 0 <= z implies x * z <= y * z;

reserve r,r1,r2 for Element of REAL+;

26 for X,Y being Subset of REAL
     st for x,y st x in X & y in Y holds x <= y
     ex z st for x,y st x in X & y in Y holds x <=z & z <= y;
28 x in NAT & y in NAT implies x + y in NAT;
29 for A being Subset of REAL
     st 0 in A & for x st x in A holds x + 1 in A
     holds NAT c= A;
30 k = { i: i < k };
```

# B.3 BOOLE

Boolean Properties of Sets by Library Committee

```
1 for X being set holds X \/ {} = X;
2 for X being set holds X /\ {} = {};
3 for X being set holds X \ {} = X;
4 for X being set holds {} \ X = {};
5 for X being set holds X \+\ {} = X;
6 for X being set st X is empty holds X = {};
7 for x, X being set st x in X holds X is non empty;
```

## B.4 XBOOLE_0

```
Boolean Properties of Sets --- Definitions
by Library Committee

reserve X, Y, Z, x, y, z for set;

scheme Separation { A()-> set, P[set] } :
ex X being set st for x being set
 holds x in X iff x in A() & P[x];

def 1 func {} -> set means not ex x being set st x in it;
der 2 func X \/ Y -> set means x in it iff x in X or x in Y;
def 3 func X /\ Y -> set means x in it iff x in X & x in Y;
def 4 func X \ Y -> set means x in it iff x in X & not x in Y;
def 5 attr X is empty means X = {};
def 6 func X \+\ Y -> set equals (X \ Y) \/ (Y \ X);
def 7 pred X misses Y means X /\ Y = {};
def 8 pred X c< Y means X c= Y & X <> Y;
def 9 pred X,Y are_c=-comparable means X c= Y or Y c= X;
def 10 redefine pred X = Y means X c= Y & Y c= X;

1 x in X \+\ Y iff not (x in X iff x in Y);
2 (for x holds not x in X iff (x in Y iff x in Z))
      implies X = Y \+\ Z;

cluster {} -> empty;
cluster empty set;
cluster non empty set;

let D be non empty set, X be set;
cluster D \/ X -> non empty;
cluster X \/ D -> non empty;

3 X meets Y iff ex x st x in X & x in Y;
4 X meets Y iff ex x st x in X /\ Y;
5 X misses Y & x in X \/ Y implies
     ((x in X & not x in Y) or (x in Y & not x inX));

scheme Extensionality { X,Y() -> set, P[set] } :
     X() = Y() provided
     for x holds x in X() iff P[x] and
     for x holds x in Y() iff P[x];

scheme SetEq { P[set] } :
     for X1,X2 being set st
    (for x being set holds x in X1 iff P[x]) &
    (for x being set holds x in X2 iff P[x]) holds X1 = X2;
```

## B.5 XBOOLE_1

```
Boolean Properties of Sets --- Theorems by Library Committee

reserve x,A,B,X,X',Y,Y',Z,V for set;

1 X c= Y & Y c= Z implies X c= Z;
2 {} c= X;
3 X c= {} implies X = {};
4 (X \/ Y) \/ Z = X \/ (Y \/ Z);
5 (X \/ Y) \/ Z = (X \/ Z) \/ (Y \/ Z);
6 X \/ (X \/ Y) = X \/ Y;
7 X c= X \/ Y;
8 X c= Z & Y c= Z implies X \/ Y c= Z;
9 X c= Y implies X \/ Z c= Y \/ Z;
10 X c= Y implies X c= Z \/ Y;
11 X \/ Y c= Z implies X c= Z;
12 X c= Y implies X \/ Y = Y;
13 X c= Y & Z c= V implies X \/ Z c= Y \/ V;
14 (Y c= X & Z c= X & for V st Y c= V & Z c= V holds X c= V) implies X
= Y \/ Z;
15 X \/ Y = {} implies X = {};
16 (X /\ Y) /\ Z = X /\ (Y /\ Z);
17 X /\ Y c= X;
18 X c= Y /\ Z implies X c= Y;
19 Z c= X & Z c= Y implies Z c= X /\ Y;
20 (X c= Y & X c= Z & for V st V c= Y & V c= Z holds V c= X) implies X
= Y /\ Z;
21 X /\ (X \/ Y) = X;
22 X \/ (X /\ Y) = X;
23 X /\ (Y \/ Z) = X /\ Y \/ X /\ Z;
24 X \/ Y /\ Z = (X \/ Y) /\ (X \/ Z);
25 (X /\ Y) \/ (Y /\ Z) \/ (Z /\ X) = (X \/ Y) /\ (Y \/ Z) /\ (Z \/ X);
26 X c= Y implies X /\ Z c= Y /\ Z;
27 X c= Y & Z c= V implies X /\ Z c= Y /\ V;
28 X c= Y implies X /\ Y = X;
29 X /\ Y c= X \/ Z;
30 X c= Z implies X \/ Y /\ Z = (X \/ Y) /\ Z;
31 (X /\ Y) \/ (X /\ Z) c= Y \/ Z;
32 X \ Y = Y \ X implies X = Y;
```

```
33 X c= Y implies X \ Z c= Y \ Z;
34 X c= Y implies Z \ Y c= Z \ X;
35 X c= Y & Z c= V implies X \ V c= Y \ Z;
36 X \ Y c= X;
37 X \ Y = {} iff X c= Y;
38 X c= Y \ X implies X = {};
39 X \/ (Y \ X) = X \/ Y;
40 (X \/ Y) \ Y = X \ Y;
41 (X \ Y) \ Z = X \ (Y \/ Z);
42 (X \/ Y) \ Z = (X \ Z) \/ (Y \ Z);
43 X c= Y \/ Z implies X \ Y c= Z;
44 X \ Y c= Z implies X c= Y \/ Z;
45 X c= Y implies Y = X \/ (Y \ X);
46 X \ (X \/ Y) = {};
47 X \ X /\ Y = X \ Y;
48 X \ (X \ Y) = X /\ Y;
49 X /\ (Y \ Z) = (X /\ Y) \ Z;
50 X /\ (Y \ Z) = X /\ Y \ X /\ Z;
51 X /\ Y \/ (X \ Y) = X;
52 X \ (Y \ Z) = (X \ Y) \/ X /\ Z;
53 X \ (Y \/ Z) = (X \ Y) /\ (X \ Z);
54 X \ (Y /\ Z) = (X \ Y) \/ (X \ Z);
55 (X \/ Y) \ (X /\ Y) = (X \ Y) \/ (Y \ X);
56 X c< Y & Y c< Z implies X c< Z;
57 not (X c< Y & Y c< X);
58 X c< Y & Y c= Z implies X c< Z;
59 X c= Y & Y c< Z implies X c< Z;
60 X c= Y implies not Y c< X;
61 X <> {} implies {} c< X;
62 not X c< {};
63 X c= Y & Y misses Z implies X misses Z;
64 A c= X & B c= Y & X misses Y implies A misses B;
65 X misses {};
66 X meets X iff X <> {};
67 X c= Y & X c= Z & Y misses Z implies X = {};
68 for A being non empty set st A c= Y & A c= Z holds Y meets Z;
69 for A being non empty set st A c= Y holds A meets Y;
70 X meets Y \/ Z iff X meets Y or X meets Z;
71 X \/ Y = Z \/ Y & X misses Y & Z misses Y implies X = Z;
72 X' \/ Y' = X \/ Y & X misses X' & Y misses Y' implies X = Y';
73 X c= Y \/ Z & X misses Z implies X c= Y;
74 X meets Y /\ Z implies X meets Y;
```

```
75 X meets Y implies X /\ Y meets Y;
76 Y misses Z implies X /\ Y misses X /\ Z;
77 X meets Y & X c= Z implies X meets Y /\ Z;
78 X misses Y implies X /\ (Y \/ Z) = X /\ Z;
79 X \ Y misses Y;
80 X misses Y implies X misses Y \ Z;
81 X misses Y \ Z implies Y misses X \ Z;
82 X \ Y misses Y \ X;
83 X misses Y iff X \ Y = X;
84 X meets Y & X misses Z implies X meets Y \ Z;
85 X c= Y implies X misses Z \ Y;
86 X c= Y & X misses Z implies X c= Y \ Z;
87 Y misses Z implies (X \ Y) \/ Z = (X \/ Z) \ Y;
88 X misses Y implies (X \/ Y) \ Y = X;
89 X /\ Y misses X \ Y;
90 X \ (X /\ Y) misses Y;
91 (X \+\ Y) \+\ Z = X \+\ (Y \+\ Z);
92 X \+\ X = {};
93 X \/ Y = (X \+\ Y) \/ X /\ Y;
94 X \/ Y = X \+\ Y \+\ X /\ Y;
95 X /\ Y = X \+\ Y \+\ (X \/ Y);
96 X \ Y c= X \+\ Y;
97 X \ Y c= Z & Y \ X c= Z implies X \+\ Y c= Z;
98 X \/ Y = X \+\ (Y \ X);
99 (X \+\ Y) \ Z = (X \ (Y \/ Z)) \/ (Y \ (X \/ Z));
100 X \ Y = X \+\ (X /\ Y);
101 X \+\ Y = (X \/ Y) \ X /\ Y;
102 X \ (Y \+\ Z) = X \ (Y \/ Z) \/ X /\ Y /\ Z;
103 X /\ Y misses X \+\ Y;
104 X c< Y or X = Y or Y c< X iff X,Y are_c=-comparable;
```

# B.6 REAL_1

```
Basic Properties of Real Numbers by Krzysztof Hryniewiecki

mode Real is Element of REAL;

reserve r for set;
reserve x,y,z,t for real number;

9 z<>0 & x*z=y*z implies x=y;
10 x + z = y + z implies x=y;

def 1 func -x -> real number means x + it = 0;
def 2 func x" -> real number means x * it = 1 if x <> 0
         otherwise it = 0;
def 3 func x-y equals x+(-y);
def 4 func x/y equals x * y";

cluster x-y -> real;
cluster x/y -> real;

redefine func -x -> Real;
redefine func x" -> Real;

redefine func x-y -> Real;
redefine func x/y -> Real;

17 x+y-z=x+(y-z);
19 0-x=-x;
21 (-x)*y = -(x*y) & (-x)*y=x*(-y);
22 x<>0 iff -x<>0;
23 x*y=0 iff x=0 or y=0;
24 x"*y"=(x*y)";
25 x-0=x;
26 -0=0;
27 x-(y+z)=x-y-z;
28 x-(y-z)=x-y+z;
29 x*(y-z)=x*y - x*z;
30 x=x+z-z;
31 x<>0 implies x"<>0;
```

```
33 1/x=x" & 1/x"=x;
34 x<>0 implies x*(1/x)=1;
35 (x/y) * (z/t) =(x*z)/(y*t);
36 x-x=0;
37 x<>0 implies x/x = 1;
38 z<>0 implies x/y=(x*z)/(y*z);
39 (-x/y=(-x)/y & x/(-y)=-x/y);
40 x/z + y/z = (x+y)/z & x/z - y/z = (x-y)/z;
41 y<>0 & t<>0 implies x/y + z/t =(x*t + z*y)/(y*t)
     & x/y - z/t =(x*t - z*y)/(y*t);
42 x/(y/z)=(x*z)/y;
43 y<>0 implies x/y*y=x;
44 for x,y ex z st x=y+z;
45 for x,y st y<>0 ex z st x=y*z;
49 x <= y implies x - z <= y - z;
50 x<=y iff -y<=-x;
52 x<=y & z<=0 implies y*z<=x*z;
53 x+z<=y+z implies x <= y;
54 x-z<=y-z implies x <= y;
55 x<=y & z<=t implies x+z<=y+t;

def 5 redefine pred x<y means x<=y & x<>y;

66 x < 0 iff 0 < -x;
67 x<y & z<=t implies x+z<y+t;
69 0<x implies y<y+x;
70 0<z & x<y implies x*z<y*z;
71 z<0 & x<y implies y*z<x*z;
72 0<z implies 0<z";
73 0<z implies (x<y iff x/z<y/z);
74 z<0 implies (x<y iff y/z<x/z);
75 x<y implies ex z st x<z & z<y;
76 for x ex y st x<y;
77 for x ex y st y<x;

scheme SepReal { P[Real]}:
ex X being Subset of REAL st
for x being Real holds x in X iff P[x];

81 (x/y)"=y/x;
82 (x/y)/(z/t)=(x*t)/(y*z);
83 -(x-y)=y-x;
```

```
84 x+y <= z iff x <= z-y;
86 x <= y+z iff x-y <= z;
92 (x <= y & z <= t implies x - t <= y - z) &
    (x < y & z <= t or x <= y & z < t impliesx-t < y-z);
93 0 <= x*x;
```

## B.7 NAT_1

```
The Fundamental Properties of Natural Numbers by Grzegorz Bancerek

mode Nat is Element of NAT;

reserve x for Real,
   k,l,m,n for Nat,
   h,i,j,p for natural number,
   X for Subset of REAL;

2 for X st 0 in X & for x st x in X holds x + 1 in X
   for k holds k in X;

redefine func n + k -> Nat;

cluster n + k -> natural;

scheme Ind { P[Nat] } :
  for k being Nat holds P[k]
  provided
  P[0] and
  for k being Nat st P[k] holds P[k + 1];

scheme Nat_Ind { P[natural number] } :
  for k being natural number holds P[k]
  provided
  P[0] and
  for k be natural number st P[k] holds P[k + 1];

redefine func n * k -> Nat;

cluster n * k -> natural;

18 0 <= i;
19 0 <> i implies 0 < i;
20 i <= j implies i * h <= j * h;
21 0 <> i + 1;
22 i = 0 or ex k st i = k + 1;
23 i + j = 0 implies i = 0 & j = 0;
```

```
scheme Def_by_Ind { N()->Nat, F(Nat,Nat)->Nat, P[Nat,Nat] } :
  (for k ex n st P[k,n] ) &
  for k,n,m st P[k,n] & P[k,m] holds n = m
  provided
  for k,n holds P[k,n] iff
  k = 0 & n = N() or ex m,l st k = m + 1 & P[m,l] & n= F(k,l);

26 for i,j st i <= j + 1 holds i <= j or i = j + 1;
27 i <= j & j <= i + 1 implies i = j or j = i + 1;
28 for i,j st i <= j ex k st j = i + k;
29 i <= i + j;

scheme Comp_Ind { P[Nat] } :
  for k holds P[k]
  provided
  for k st for n st n < k holds P[n] holds P[k];

scheme Min { P[Nat] } :
  ex k st P[k] & for n st P[n] holds k <= n
  provided
  ex k st P[k];

scheme Max { P[Nat],N()->Nat } :
  ex k st P[k] & for n st P[n] holds n <= k
  provided
  for k st P[k] holds k <= N() and
  ex k st P[k];

37 i <= j implies i <= j + h;
38 i < j + 1 iff i <= j;
40 i * j = 1 implies i = 1 & j = 1;

scheme Regr { P[Nat] } :
  P[0]
  provided
  ex k st P[k] and
  for k st k <> 0 & P[k] ex n st n < k & P[n];

reserve k1,t,t1 for Nat;

42 for m st 0 < m for n ex k,t st n = (m*k)+t & t < m;
```

```
43 for n,m,k,k1,t,t1 being natural number
st n = m*k+t & t < m & n = m*k1+t1 & t1 < m holds
k = k1 & t = t1;

def 1 func k div l -> Nat means
 ( ex t st k = l * it + t & t < l ) or it = 0 & l = 0;
def 2 func k mod l -> Nat means
 ( ex t st k = l * t + it & it < l ) or it = 0 & l = 0;

46 0 < i implies j mod i < i;
47 0 < i implies j = i * (j div i) + (j mod i);

def 3 pred k divides l means ex t st l = k * t;

49 j divides i iff i = j * (i div j);
51 i divides j & j divides h implies i divides h;
52 i divides j & j divides i implies i = j;
53 i divides 0 & 1 divides i;
54 0 < j & i divides j implies i <= j;
55 i divides j & i divides h implies i divides j+h;
56 i divides j implies i divides j * h;
57 i divides j & i divides j + h implies i divides h;
58 i divides j & i divides h implies i divides j mod h;

def 4 func k lcm n -> Nat means
  k divides it & n divides it & for m st k divides m &n divides
  m holds it divides m;
def 5 func k hcf n -> Nat means
  it divides k & it divides n & for m st m divides k &m divides
  n holds m divides it;

scheme Euklides { Q(Nat)->Nat, a,b()->Nat } :
  ex n st Q(n) = a() hcf b() & Q(n + 1) = 0
  provided
  0 < b() & b() < a() and
  Q(0) = a() & Q(1) = b() and
  for n holds Q(n + 2) = Q(n) mod Q(n + 1);

cluster -> ordinal Nat;
cluster non empty ordinal Subset of REAL;
```

# B.8 FUNCT_1

```
Functions and Their Basic Properties by Czeslaw Bylinski

reserve X,X1,X2,Y,Y1,Y2 for set,
p,x,x1,x2,y,y1,y2,z,z1,z2 for set;

def 1 attr X is Function-like means
      for x,y1,y2 st [x,y1] in X & [x,y2] in X holdsy1 = y2;

cluster Relation-like Function-like set;

mode Function is Function-like Relation-like set;

cluster empty -> Function-like set;

reserve f,f1,f2,g,g1,g2,h for Function;

2 for F being set st
    (for p st p in F ex x,y st [x,y] = p) &
    (for x,y1,y2 st [x,y1] in F & [x,y2] in F holds y1 =y2)
    holds F is Function;

scheme GraphFunc{A()->set,P[set,set]}:
 ex f st for x,y holds [x,y] in f iff x in A() & P[x,y]
 provided
 for x,y1,y2 st P[x,y1] & P[x,y2] holds y1 = y2;

def 4 func f.x -> set means
     [x,it] in f if x in dom f otherwise it = {};

8 [x,y] in f iff x in dom f & y = f.x;
9 dom f = dom g & (for x st x in dom f holds f.x = g.x) implies f = g;

def 5 redefine func rng f means
     for y holds y in it iff ex x st x in dom f & y= f.x;

12 x in dom f implies f.x in rng f;
14 dom f = {x} implies rng f = {f.x};
```

```
scheme FuncEx{A()->set,P[set,set]}:
 ex f st dom f = A() & for x st x in A() holds P[x,f.x]
 provided
 for x,y1,y2 st x in A() & P[x,y1] & P[x,y2] holds y1 = y2 and
 for x st x in A() ex y st P[x,y];

scheme Lambda{A()->set,F(set)->set}:
 ex f being Function st dom f = A() & for x st x in A()
 holds f.x =  F(x);

15 X <> {} implies for y ex f st dom f = X & rng f = {y};
16 (for f,g st dom f = X & dom g = X holds f = g) implies X = {};
17 dom f = dom g & rng f = {y} & rng g = {y} implies f = g;
18 Y <> {} or X = {} implies ex f st X = dom f & rng f c= Y;
19 (for y st y in Y ex x st x in dom f & y = f.x) implies Y c= rng f;

redefine func f*g;
synonym g*f;

cluster g*f -> Function-like;

20 for h st
    (for x holds x in dom h iff x in dom f & f.x in dom g)&
    (for x st x in dom h holds h.x = g.(f.x))
    holds h = g*f;

21 x in dom(g*f) iff x in dom f & f.x in dom g;
22 x in dom(g*f) implies (g*f).x = g.(f.x);
23 x in dom f implies (g*f).x = g.(f.x);

25 z in rng(g*f) implies z in rng g;

27 dom(g*f) = dom f implies rng f c= dom g;
33 rng f c= Y & (for g,h st dom g = Y & dom h = Y & g*f = h*f
     holds g = h) implies Y = rng f;

redefine func diagonal X;
synonym id X;

cluster id X -> Function-like;

34 f = id X iff dom f = X & for x st x in X holds f.x = x;
```

```
35 x in X implies (id X).x = x;
37 dom(f*(id X)) = dom f /\ X;
38 x in dom f /\ X implies f.x = (f*(id X)).x;
40 x in dom((id Y)*f) iff x in dom f & f.x in Y;
42 f*(id dom f) = f & (id rng f)*f = f;
43 (id X)*(id Y) = id(X /\ Y);
44 rng f = dom g & g*f = f implies g = id dom g;

def 8 attr f is one-to-one means
    for x1,x2 st x1 in dom f & x2 in dom f & f.x1 = f.x2holds x1 = x2;

46 f is one-to-one & g is one-to-one implies g*f is one-to-one;
47 g*f is one-to-one & rng f c= dom g implies f is one-to-one;
48 g*f is one-to-one & rng f = dom g implies f is one-to-one &
     g is one-to-one;
49 f is one-to-one iff
     (for g,h st rng g c= dom f & rng h c= dom f &dom g = dom h &
      f*g = f*h holds g = h);
50 dom f = X & dom g = X & rng g c= X & f is one-to-one & f*g = f
     implies g = id X;
51 rng(g*f) = rng g & g is one-to-one implies dom g c= rng f;
52 id X is one-to-one;
53 (ex g st g*f = id dom f) implies f is one-to-one;

cluster empty Function;

cluster empty -> one-to-one Function;

cluster one-to-one Function;

cluster f~ -> Function-like;

def 9 func f" -> Function equals f~;

54 f is one-to-one implies for g being Function holds g=f" iff
    dom g = rng f &for y,x holds y in rng f & x = g.y
    iff x in dom f & y = f.x;
55 f is one-to-one implies rng f = dom(f") & dom f = rng(f");
56 f is one-to-one & x in dom f implies x = (f").(f.x) & x = (f"*f).x;
57 f is one-to-one & y in rng f implies y = f.((f").y) & y = (f*f").y;
58 f is one-to-one implies dom(f"*f) = dom f & rng(f"*f) = dom f;
59 f is one-to-one implies dom(f*f") = rng f & rng(f*f") = rng f;
```

```
60 f is one-to-one & dom f = rng g & rng f = dom g &
    (for x,y st x in dom f & y in dom g holds f.x = y iffg.y = x)
    implies g = f";
61 f is one-to-one implies f"*f = id dom f & f*f" = id rng f;
62 f is one-to-one implies f" is one-to-one;
63 f is one-to-one & rng f = dom g & g*f = id dom f implies g = f";
64 f is one-to-one & rng g = dom f & f*g = id rng f implies g = f";
65 f is one-to-one implies (f")" = f;
66 f is one-to-one & g is one-to-one implies (g*f)" = f"*g";
67 (id X)" = (id X);

cluster f|X -> Function-like;

68 g = f|X iff dom g = dom f /\ X & for x st x in dom g
    holds g.x = f.x;

70 x in dom(f|X) implies (f|X).x = f.x;
71 x in dom f /\ X implies (f|X).x = f.x;
72 x in X implies (f|X).x = f.x;
73 x in dom f & x in X implies f.x in rng(f|X);
74 X c= dom f implies dom(f|X) = X;
76 dom(f|X) c= dom f & rng(f|X) c= rng f;
82 X c= Y implies (f|X)|Y = f|X & (f|Y)|X = f|X;
84 f is one-to-one implies f|X is one-to-one;

cluster Y|f -> Function-like;

85 g = Y|f iff (for x holds x in dom g iff x in dom f & f.x in Y) &
    (for x st x in dom g holds g.x = f.x);
86 x in dom(Y|f) iff x in dom f & f.x in Y;
87 x in dom(Y|f) implies (Y|f).x = f.x;
89 dom(Y|f) c= dom f & rng(Y|f) c= rng f;
97 X c= Y implies Y|(X|f) = X|f & X|(Y|f) = X|f;
99 f is one-to-one implies Y|f is one-to-one;

def 12 func f.:X means
     for y holds y in it iff ex x st x in dom f & xin X & y = f.x;

117 x in dom f implies f.:{x} = {f.x};
118 x1 in dom f & x2 in dom f implies f.:{x1,x2} = {f.x1,f.x2};
120 (Y|f).:X c= f.:X;
121 f is one-to-one implies f.:(X1 /\ X2) = f.:X1 /\ f.:X2;
```

```
122 (for X1,X2 holds f.:(X1 /\ X2) = f.:X1 /\ f.:X2)
      implies f is one-to-one;
123 f is one-to-one implies f.:(X1 \ X2) = f.:X1 \ f.:X2;
124 (for X1,X2 holds f.:(X1 \ X2) = f.:X1 \ f.:X2)
      implies f is one-to-one;
125 X misses Y & f is one-to-one implies f.:X misses f.:Y;
126 (Y|f).:X = Y /\ f.:X;

def 13 redefine func f"Y means
      for x holds x in it iff x in dom f & f.x in Y;

137 f"(Y1 /\ Y2) = f"Y1 /\ f"Y2;
138 f"(Y1 \ Y2) = f"Y1 \ f"Y2;
139 (f|X)"Y = X /\ (f"Y);
142 y in rng f iff f"{y} <> {};
143 (for y st y in Y holds f"{y} <> {}) implies Y c= rng f;
144 (for y st y in rng f ex x st f"{y} = {x}) iff f is one-to-one;
145 f.:(f"Y) c= Y;
146 X c= dom f implies X c= f"(f.:X);
147 Y c= rng f implies f.:(f"Y) = Y;
148 f.:(f"Y) = Y /\ f.:(dom f);
149 f.:(X /\ f"Y) c= (f.:X) /\ Y;
150 f.:(X /\ f"Y) = (f.:X) /\ Y;
151 X /\ f"Y c= f"(f.:X /\ Y);
152 f is one-to-one implies f"(f.:X) c= X;
153 (for X holds f"(f.:X) c= X) implies f is one-to-one;
154 f is one-to-one implies f.:X = (f")"X;
155 f is one-to-one implies f"Y = (f").:Y;
156 Y = rng f & dom g = Y & dom h = Y & g*f = h*f implies g = h;
157 f.:X1 c= f.:X2 & X1 c= dom f & f is one-to-one implies X1 c= X2;
158 f"Y1 c= f"Y2 & Y1 c= rng f implies Y1 c= Y2;
159 f is one-to-one iff for y ex x st f"{y} c= {x};
160 rng f c= dom g implies f"X c= (g*f)"(g.:X);
```

## B.9 SUBSET_1

```
Properties of Subsets by Zinaida Trybulec

reserve E,X,x,y for set;

cluster bool X -> non empty;
cluster { x } -> non empty;
cluster { x, y } -> non empty;

def 2 mode Element of X means
     it in X if X is non empty otherwise it is empty;

mode Subset of X is Element of bool X;

cluster non empty Subset of X;

cluster [: X1,X2 :] -> non empty;

cluster [: X1,X2,X3 :] -> non empty;

cluster [: X1,X2,X3,X4 :] -> non empty;

redefine mode Element of X -> Element of D;

cluster empty Subset of E;

def 3 func {} E -> empty Subset of E equals {};
def 4 func [#] E -> Subset of E equals E;

4 {} is Subset of X;

reserve A,B,C for Subset of E;

7 (for x being Element of E holds x in A implies x in B)
    implies A c= B;
8 (for x being Element of E holds x in A iff x in B) implies A = B;
10 A <> {} implies ex x being Element of E st x in A;

def 5 func A` -> Subset of E equals E \ A;
```

```
redefine func A \/ B -> Subset of E;
func A /\ B -> Subset of E;
func A \ B -> Subset of E;
func A \+\ B -> Subset of E;

15 (for x being Element of E holds x in A iff x in B or x in C)
     implies A = B \/ C;
16 (for x being Element of E holds x in A iff x in B & x in C)
     implies A = B /\ C;
17 (for x being Element of E holds x in A iff x in B & not x in C)
     implies A = B \ C;
18 (for x being Element of E holds x in A iff not(x in B iff x in C))
     implies A = B \+\ C;
21 {} E = ([#] E)`;
22 [#] E = ({} E)`;
25 A \/ A` = [#]E;
26 A misses A`;
28 A \/ [#]E = [#]E;
29 (A \/ B)` = A` /\ B`;
30 (A /\ B)` = A` \/ B`;
31 A c= B iff B` c= A`;
32 A \ B = A /\ B`;
33 (A \ B)` = A` \/ B;
34 (A \+\ B)` = A /\ B \/ A` /\ B`;
35 A c= B` implies B c= A`;
36 A` c= B implies B` c= A;
38 A c= A` iff A = {}E;
39 A` c= A iff A = [#]E;
40 X c= A & X c= A` implies X = {};
41 (A \/ B)` c= A`;
42 A` c= (A /\ B)`;
43 A misses B iff A c= B`;
44 A misses B` iff A c= B;
46 A misses B & A` misses B` implies A = B`;
47 A c= B & C misses B implies A c= C`;
48 (for a being Element of A holds a in B) implies A c= B;
49 (for x being Element of E holds x in A) implies E = A;
50 E <> {} implies for A,B holds A = B` iff
    for x being Element of E holds x in A iff not x in B;
51 E <> {} implies for A,B holds A = B` iff
    for x being Element of E holds not x in A iff x in B;
52 E <> {} implies for A,B holds A = B` iff
```

```
    for x being Element of E holds not(x in A iff x in B);
53 x in A` implies not x in A;

reserve x1,x2,x3,x4,x5,x6,x7,x8 for Element of X;

54 X <> {} implies {x1} is Subset of X;
55 X <> {} implies {x1,x2} is Subset of X;
56 X <> {} implies {x1,x2,x3} is Subset of X;
57 X <> {} implies {x1,x2,x3,x4} is Subset of X;
58 X <> {} implies {x1,x2,x3,x4,x5} is Subset of X;
59 X <> {} implies {x1,x2,x3,x4,x5,x6} is Subset of X;
60 X <> {} implies {x1,x2,x3,x4,x5,x6,x7} is Subset of X;
61 X <> {} implies {x1,x2,x3,x4,x5,x6,x7,x8} is Subset of X;
62 x in X implies {x} is Subset of X;

scheme Subset_Ex { A()-> set, P[set] } :
  ex X being Subset of A() st for x
  holds x in X iff x in A() & P[x];

scheme Subset_Eq {X() -> set, P[set]}:
 for X1,X2 being Subset of X() st
  (for y being Element of X() holds y in X1 iff P[y]) &
  (for y being Element of X() holds y in X2 iff P[y])
  holds X1 = X2;

redefine pred X misses Y;
```

# B.10 FINSEQ_1

```
Segments of Natural Numbers and Finite Sequences
by Grzegorz Bancerek, and Krzysztof Hryniewiecki

reserve k,l,m,n,k1,k2 for Nat,
           a,b,c for natural number,
           x,y,z,y1,y2,X,Y for set,
           f,g for Function;

def 1 func Seg n -> set equals { k : 1 <= k & k <= n };
redefine func Seg n -> Subset of NAT;

3 a in Seg b iff 1 <= a & a <= b;
4 Seg 0 = {} & Seg 1 = { 1 } & Seg 2 = { 1,2 };
5 a = 0 or a in Seg a;
6 a+1 in Seg(a+1);
7 a <= b iff Seg a c= Seg b;
8 Seg a = Seg b implies a = b;
9 c <= a implies
     Seg c = Seg c /\ Seg a & Seg c = Seg a /\ Seg c;
10 (Seg c = Seg c /\ Seg a or Seg c = Seg a /\ Seg c )
      implies c <= a;
11 Seg a \/ { a+1 } = Seg (a+1);

def 2 attr IT is FinSequence-like means ex n st dom IT = Seg n;
     cluster FinSequence-like Function;
     mode FinSequence is FinSequence-like Function;

reserve p,q,r,s,t for FinSequence;

cluster Seg n -> finite;
cluster FinSequence-like -> finite Function;

def 3 func Card p -> Nat means Seg it = dom p;
     redefine func dom p -> Subset of NAT;

14 {} is FinSequence;
15 (ex k st dom f c= Seg k) implies ex p st f c= p;
```

```
scheme SeqEx{A()->Nat,P[set,set]}:
  ex p st dom p = Seg A() & for k st k in Seg A() holds P[k,p.k]
  provided
  for k,y1,y2 st k in Seg A() & P[k,y1] & P[k,y2] holds y1=y2
   and
  for k st k in Seg A() ex x st P[k,x];

scheme SeqLambda{A()->Nat,F(set) -> set}:
  ex p being FinSequence st len p = A() & for k st k in Seg A()
  holds p.k=F(k);

16 z in p implies ex k st k in dom p & z=[k,p.k];
17 X = dom p & X = dom q & (for k st k in X holds p.k = q.k)
   implies p=q;
18 ( (len p = len q) & for k st 1 <=k & k <= len p holds p.k=q.k )
    implies p=q;
19 p|(Seg a) is FinSequence;
20 rng p c= dom f implies f*p is FinSequence;
21 a <= len p & q = p|(Seg a) implies len q = a & dom q = Seg a;

def 4 mode FinSequence of D -> FinSequence means rng it c= D;
     cluster {} -> FinSequence-like;
     cluster FinSequence-like PartFunc of NAT,D;

redefine mode FinSequence of D -> FinSequence-like PartFunc of NAT,D;

reserve D for set;

23 for p being FinSequence of D holds p|(Seg a) is FinSequence of D;
24 for D being non empty set
    ex p being FinSequence of D st len p = a;

cluster empty FinSequence;

25 len p = 0 iff p = {};
26 p={} iff dom p = {};
27 p={} iff rng p= {};
29 for D be set holds {} is FinSequence of D;

cluster empty FinSequence of D;

def 5 func <*x*> -> set equals { [1,x] };
def 6 func <*>D -> empty FinSequence of D equals {};

32 p=<*>(D) iff len p = 0;

def 7 func p^q -> FinSequence means
     dom it = Seg (len p + len q) &
     (for k st k in dom p holds it.k=p.k) &
     (for k st k in dom q holds it.(len p + k) = q.k);

35 len(p^q) = len p + len q;
36 (len p + 1 <= k & k <= len p + len q) implies (p^q).k=q.(k-len p);
37 len p < k & k <= len(p^q) implies (p^q).k = q.(k - len p);
38 k in dom (p^q) implies
(k in dom p or (ex n st n in dom q & k=len p + n));
39 dom p c= dom(p^q);
40 x in dom q implies ex k st k=x & len p + k in dom(p^q);
41 k in dom q implies len p + k in dom(p^q);
42 rng p c= rng(p^q);
43 rng q c= rng(p^q);
44 rng(p^q) = rng p \/ rng q;
45 p^q^r = p^(q^r);
46 p^r = q^r or r^p = r^q implies p = q;
47 p^{} = p & {}^p = p;
48 p^q = {} implies p={} & q={};

redefine func p^q -> FinSequence of D;

def 8 redefine func <*x*> -> Function means dom it = Seg 1 & it.1 = x;
     cluster <*x*> -> Function-like Relation-like;
     cluster <*x*> -> FinSequence-like;

50 p^q is FinSequence of D implies
     p is FinSequence of D & q is FinSequence of D;

def 9 func <*x,y*> -> set equals <*x*>^<*y*>;
def 10 func <*x,y,z*> -> set equals <*x*>^<*y*>^<*z*>;

cluster <*x,y*> -> Function-like Relation-like;
cluster <*x,y,z*> -> Function-like Relation-like;
cluster <*x,y*> -> FinSequence-like;
cluster <*x,y,z*> -> FinSequence-like;
```

```
52 <*x*> = { [1,x] };
55 p=<*x*> iff dom p = Seg 1 & rng p = {x};
56 p=<*x*> iff len p = 1 & rng p = {x};
57 p = <*x*> iff len p = 1 & p.1 = x;
58 (<*x*>^p).1 = x;
59 (p^<*x*>).(len p + 1)=x;
60 <*x,y,z*>=<*x*>^<*y,z*> &
    <*x,y,z*>=<*x,y*>^<*z*>;
61 p = <*x,y*> iff len p = 2 & p.1=x & p.2=y;
62 p = <*x,y,z*> iff len p = 3 & p.1 = x & p.2 = y & p.3 = z;
63 p <> {} implies ex q,x st p=q^<*x*>;

redefine func <*x*> -> FinSequence of D;

scheme IndSeq{P[FinSequence]}:
 for p holds P[p]
   provided
   P[{}] and
   for p,x st P[p] holds P[p^<*x*>];

64 for p,q,r,s being FinSequence st p^q = r^s & len p <= len r
     ex t being FinSequence st p^t = r;

def 11 func D* -> set means x in it iff x is FinSequence of D;

cluster D* -> non empty;

66 {} in D*;

scheme SepSeq{D()->non empty set, P[FinSequence]}:
  ex X st (for x holds x in X iff
    ex p st (p in D()* & P[p] & x=p));

def 12 attr IT is FinSubsequence-like means ex k st dom IT c= Seg k;

cluster FinSubsequence-like Function;
mode FinSubsequence is FinSubsequence-like Function;

68 for p being FinSequence holds p is FinSubsequence;
69 p|X is FinSubsequence & X|p is FinSubsequence;

reserve p' for FinSubsequence;
```

```
def 13 given k such that X c= Seg k;
     func Sgm X -> FinSequence of NAT means
     rng it = X & for l,m,k1,k2 st
       ( 1 <= l & l < m & m <= lenit &
           k1=it.l & k2=it.m) holds k1< k2;

71 rng Sgm dom p' = dom p';

def 14 func Seq p' -> Function equals p'* Sgm(dom p');

cluster Seq p' -> FinSequence-like;

72 for X st ex k st X c= Seg k holds Sgm X = {} iff X = {};
73 D is finite iff ex p st D = rng p;

cluster rng p -> finite;

74 Seg n,Seg m are_equipotent implies n = m;
75 Seg n,n are_equipotent;
76 Card Seg n = Card n;
77 X is finite implies ex n st X,Seg n are_equipotent;
78 for n being Nat holds
     card Seg n = n & card n = n & card Card n =n;
```

# B.11 FUNCT_2

Functions from a Set to a Set by Czeslaw Bylinski

```
reserve P,Q,X,Y,Y1,Y2,Z,p,x,x',x1,x2,y,y1,y2,z for set;

def 1 attr R is quasi_total means X = dom R if Y = {} implies X = {}
otherwise R = {};

cluster quasi_total Function-like Relation of X,Y;
cluster total -> quasi_total PartFunc of X,Y;
mode Function of X,Y is quasi_total Function-like Relation of X,Y;

3 for f being Function holds f is Function of dom f, rng f;
4 for f being Function st rng f c= Y holds f is Function of dom f, Y;
5 for f being Function st dom f = X & for x st x in X
    holds f.x in Y holds f is Function of X,Y;
6 for f being Function of X,Y st Y <> {} & x in X holds f.x in rng f;
7 for f being Function of X,Y st Y <> {} & x in X holds f.x in Y;
8 for f being Function of X,Y st (Y = {} implies X = {}) & rng f c=Z
     holds f is Function of X,Z;
9 for f being Function of X,Y
st (Y = {} implies X = {}) & Y c= Z holds f is Function of X,Z;

scheme FuncEx1{X, Y() -> set, P[set,set]}:
ex f being Function of X(),Y() st for x st x in X() holds P[x,f.x]
provided
for x st x in X() ex y st y in Y() & P[x,y];

scheme Lambda1{X, Y() -> set, F(set)->set}:
ex f being Function of X(),Y() st for x st x in X() holds f.x = F(x)
provided
for x st x in X() holds F(x) in Y();

def 2 func Funcs(X,Y) -> set means
     x in it iff ex f being Function st x = f & domf = X & rng f c= Y;

11 for f being Function of X,Y st Y = {} implies X = {}
      holds f in Funcs(X,Y);
12 for f being Function of X,X holds f in Funcs(X,X);
```

```
14 X <> {} implies Funcs(X,{}) = {};
16 for f being Function of X,Y
     st Y <> {} & for y st y in Y ex x st x inX & y = f.x
     holds rng f = Y;
17 for f being Function of X,Y st y in Y &
     rng f = Y ex x st x in X & f.x = y;
18 for f1,f2 being Function of X,Y
      st for x st x in X holds f1.x = f2.x
      holds f1 = f2;
19 for f being Function of X,Y for g being Function of Y,Z
      st Y = {} implies Z = {} or X = {}
      holds g*f is Function of X,Z;
20 for f being Function of X,Y for g being Function of Y,Z
     st Y <> {} & Z <> {} & rng f =Y & rng g = Z holds rng(g*f) = Z;
21 for f being Function of X,Y, g being Function
     st Y <> {} & x in X holds (g*f).x = g.(f.x);
22 for f being Function of X,Y st Y <> {} holds rng f = Y iff
     for Z st Z <> {} for g,h being Function of Y,Zst g*f = h*f
      holds g = h;
23 for f being Function of X,Y
     st Y = {} implies X = {} holds f*(id X) = f & (idY)*f = f;
24 for f being Function of X,Y for g being Function of Y,X
      st f*g = id Y holds rng f = Y;
25 for f being Function of X,Y st Y = {} implies X = {}
     holds f is one-to-one iff
     for x1,x2 st x1 in X & x2 in X & f.x1 = f.x2holds x1 = x2;
     for f being Function of X,Y for g being Function ofY,Z
     st (Z = {} implies Y = {}) & (Y = {}
     implies X = {}) & g*f is one-to-one
     holds f is one-to-one;
27 for f being Function of X,Y st X <> {} & Y <> {}
     holds f is one-to-one iff
     for Z for g,h being Function of Z,X st f*g = f*h holdsg = h;
28 for f being Function of X,Y for g being Function of Y,Z
     st Z <> {} & rng(g*f) = Z & g is one-to-oneholds rng f = Y;
29 for f being Function of X,Y for g being Function of Y,X
     st Y <> {} & g*f = id X holds f is one-to-one& rng g = X;
30 for f being Function of X,Y for g being Function of Y,Z
     st (Z = {} implies Y = {}) & g*f is one-to-one& rng f = Y
     holds f is one-to-one & g is one-to-one;
31 for f being Function of X,Y st f is one-to-one & rng f = Y
     holds f" is Function of Y,X;
```

```
32 for f being Function of X,Y
     st Y <> {} & f is one-to-one & x in Xholds (f").(f.x) = x;
34 for f being Function of X,Y for g being Function of Y,X
     st X <> {} & Y <> {} & rng f =Y & f is one-to-one &
     for y,x holds y in Y & g.y = x iff x in X &f.x = y
     holds g = f";
35 for f being Function of X,Y
     st Y <> {} & rng f = Y & f is one-to-one
     holds f"*f = id X & f*f" = id Y;
36 for f being Function of X,Y for g being Function of Y,X
     st X <> {} & Y <> {} & rng f =Y & g*f = id X & f is one-to-one
     holds g = f";
37 for f being Function of X,Y st Y <> {}
     & ex g being Function of Y,X st g*f = id X
      holds f is one-to-one;
38 for f being Function of X,Y
      st (Y = {} implies X = {}) & Z c= X holds f|Zis Function of Z,Y;
40 for f being Function of X,Y st X c= Z holds f|Z = f;
41 for f being Function of X,Y st Y <> {} & x in X & f.x in
     Z holds (Z|f).x = f.x;
42 for f being Function of X,Y st (Y = {} implies X = {}) & Y c=
Z         holds Z|f = f;
43 for f being Function of X,Y st Y <> {}
     for y holds y in f.:P iff ex x st x in X & x inP & y = f.x;
44 for f being Function of X,Y holds f.:P c= Y;

redefine func f.:P -> Subset of Y;

45 for f being Function of X,Y st Y = {} implies X = {}
     holds f.:X = rng f;
46 for f being Function of X,Y
     st Y <> {} for x holds x in f"Q iff x in X & f.x in Q;
47 for f being PartFunc of X,Y holds f"Q c= X;

redefine func f"Q -> Subset of X;

48 for f being Function of X,Y st Y = {} implies X = {}
     holds f"Y = X;
49 for f being Function of X,Y
     holds (for y st y in Y holds f"{y} <> {})iff rng f = Y;
50 for f being Function of X,Y
      st (Y = {} implies X = {}) & P c= X holds P c=f"(f.:P);
```

```
51 for f being Function of X,Y st Y = {} implies X = {}
     holds f"(f.:X) = X;
53 for f being Function of X,Y for g being Function of Y,Z
     st (Z = {} implies Y = {}) & (Y = {} implies X= {})
     holds f"Q c= (g*f)"(g.:Q);
54 for f being Function of {},Y holds dom f = {} & rng f = {};
55 for f being Function st dom f = {} holds f is Function of {},Y;
56 for f1 being Function of {},Y1 for f2 being Function of {},Y2
     holds f1 = f2;
58 for f being Function of {},Y holds f is one-to-one;
59 for f being Function of {},Y holds f.:P = {};
60 for f being Function of {},Y holds f"Q = {};
61 for f being Function of {x},Y st Y <> {} holds f.x in Y;
62 for f being Function of {x},Y st Y <> {} holds rng f = {f.x};
63 for f being Function of {x},Y st Y <> {} holds f is one-to-one;
64 for f being Function of {x},Y st Y <> {} holds f.:P c= {f.x};
65 for f being Function of X,{y} st x in X holds f.x = y;
66 for f1,f2 being Function of X,{y} holds f1 = f2;

redefine func g*f -> Function of X,X;
redefine func id X -> Function of X,X;

67 for f being Function of X,X holds dom f = X & rng f c= X;
70 for f being Function of X,X, g being Function
st x in X holds (g*f).x = g.(f.x);
73 for f,g being Function of X,X st rng f = X & rng g = X
     holds rng(g*f) = X;
74 for f being Function of X,X holds f*(id X) = f & (id X)*f = f;
75 for f,g being Function of X,X st g*f = f & rng f = X
     holds g = id X;
76 for f,g being Function of X,X st f*g = f & f is one-to-one
     holds g = id X;
77 for f being Function of X,X holds f is one-to-one iff
     for x1,x2 st x1 in X & x2 in X & f.x1 = f.x2holds x1 = x2;
79 for f being Function of X,X holds f.:X = rng f;
82 for f being Function of X,X holds f"(f.:X) = X;

def 3 attr f is onto means rng f = Y;
def 4 attr f is bijective means f is one-to-one onto;

cluster bijective -> one-to-one onto Function of X,Y;
cluster one-to-one onto -> bijective Function of X,Y;
cluster bijective Function of X,X;

mode Permutation of X is bijective Function of X,X;

83 for f being Function of X, X holds
     f is Permutation of X iff f is one-to-one & rngf = X;
85 for f being Function of X,X st f is one-to-one holds
     for x1,x2 st x1 in X & x2 in X & f.x1 = f.x2holds x1 = x2;

redefine func g*f -> Permutation of X;

redefine func id X -> Permutation of X;

redefine func f" -> Permutation of X;

86 for f,g being Permutation of X st g*f = g holds f = id X;
87 for f,g being Permutation of X st g*f = id X holds g = f";
88 for f being Permutation of X holds (f")*f =id X & f*(f") = id X;
92 for f being Permutation of X st P c= X
     holds f.:(f"P) = P & f"(f.:P) = P;
93 for f being Function of X,X st f is one-to-one
     holds f.:P = (f")"P & f"P = (f").:P;

reserve C,D,E for non empty set;

cluster quasi_total -> total PartFunc of X,D;

redefine func g*f -> Function of X,Z;

reserve c for Element of C;
reserve d for Element of D;

redefine func f.c -> Element of D;

scheme FuncExD{C, D() -> non empty set, P[set,set]}:
ex f being Function of C(),D() st for x being Element of C()
holds P[x,f.x]
provided
for x being Element of C() ex y being Element of D() st P[x,y];

scheme LambdaD{C, D() -> non empty set,
F(Element of C()) -> Element of D()}:
ex f being Function of C(),D() st
for x being Element of C() holds f.x = F(x);

113 for f1,f2 being Function of X,Y st
       for x being Element of X holds f1.x = f2.x holds f1 = f2;
116 for f being Function of C,D for d
       holds d in f.:P iff ex c st c in P & d =f.c;
118 for f1,f2 being Function of [:X,Y:],Z
       st for x,y st x in X & y in Y holds f1.[x,y]= f2.[x,y]
       holds f1 = f2;
119 for f being Function of [:X,Y:],Z st x in X & y in Y & Z <> {}
       holds f.[x,y] in Z;

scheme FuncEx2{X, Y, Z() -> set, P[set,set,set]}:
ex f being Function of [:X(),Y():],Z() st
for x,y st x in X() & y in Y() holds P[x,y,f.[x,y]]
provided
for x,y st x in X() & y in Y() ex z st z in Z() & P[x,y,z];

scheme Lambda2{X, Y, Z() -> set, F(set,set)->set}:
ex f being Function of [:X(),Y():],Z()
st for x,y st x in X() & y in Y() holds f.[x,y] = F(x,y)
provided
for x,y st x in X() & y in Y() holds F(x,y) in Z();

120 for f1,f2 being Function of [:C,D:],E st for c,d
       holds f1.[c,d] = f2.[c,d] holds f1 = f2;

scheme FuncEx2D{X, Y, Z() -> non empty set, P[set,set,set]}:
ex f being Function of [:X(),Y():],Z() st
for x being Element of X() for y being Element of Y() holds
P[x,y,f.[x,y]]
provided
for x being Element of X() for y being Element of Y()
ex z being Element of Z() st P[x,y,z];

scheme 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);
```

```
121 for f being set st f in Funcs(X,Y) holds f is Function of X,Y;

scheme Lambda1C{A, B() -> set, C[set], F(set)->set, G(set)->set}:
ex f being Function of A(),B() st
for x st x in A() holds
(C[x] implies f.x = F(x)) & (not C[x] implies f.x = G(x))
provided
for x st x in A() holds
(C[x] implies F(x) in B()) & (not C[x] implies G(x) in B());

123 for f being Function of {},Y holds f = {};
124 for f being Function of X,Y st f is one-to-one
      holds f" is PartFunc of Y,X;
125 for f being Function of X,X st f is one-to-one
      holds f" is PartFunc of X,X;
127 for f being Function of X,Y st Y = {}
      implies X = {} holds <:f,X,Y:> = f;
128 for f being Function of X,X holds <:f,X,X:> = f;
130 for f being PartFunc of X,Y st dom f = X
      holds f is Function of X,Y;
131 for f being PartFunc of X,Y st f is total
      holds f is Function of X,Y;
132 for f being PartFunc of X,Y st (Y = {}
      implies X = {}) & f is Function of X,Y holdsf is total;
133 for f being Function of X,Y
      st (Y = {} implies X = {}) holds <:f,X,Y:> is total;
134 for f being Function of X,X holds <:f,X,X:> is total;
136 for f being PartFunc of X,Y st Y = {} implies X = {}
      ex g being Function of X,Y st for x st x in dom f
      holds g.x = f.x;
141 Funcs(X,Y) c= PFuncs(X,Y);
142 for f,g being Function of X,Y st (Y = {}
      implies X = {}) & f tolerates g holds f =g;
143 for f,g being Function of X,X st f tolerates g holds f = g;
145 for f being PartFunc of X,Y for g being Function of X,Y
      st Y = {} implies X = {}
      holds f tolerates g iff for x st x in dom f holdsf.x = g.x;
146 for f being PartFunc of X,X for g being Function of X,X
      holds f tolerates g iff for x st x in dom f holds f.x = g.x;
148 for f being PartFunc of X,Y st Y = {} implies X = {}
      ex g being Function of X,Y st f tolerates g;
```

```
149 for f being PartFunc of X,X ex g being Function of X,X st
      f tolerates g;
151 for f,g being PartFunc of X,Y for h being Function of X,Y
      st (Y = {} implies X = {}) & f tolerates h & g tolerates h
      holds f tolerates g;
152 for f,g being PartFunc of X,X for h being Function of X,X
      st f tolerates h & g tolerates h holds ftolerates g;
154 for f,g being PartFunc of X,Y st (Y = {}
      implies X = {}) & f tolerates g
      ex h being Function of X,Y st f tolerates h &g tolerates h;
155 for f being PartFunc of X,Y for g being Function of X,Y
      st (Y = {} implies X = {}) & f toleratesg
      holds g in TotFuncs f;
156 for f being PartFunc of X,X for g being Function of X,X
      st f tolerates g holds g in TotFuncs f;
158 for f being PartFunc of X,Y for g being set
      st g in TotFuncs(f) holds g is Function of X,Y;
159 for f being PartFunc of X,Y holds TotFuncs f c= Funcs(X,Y);
160 TotFuncs <:{},X,Y:> = Funcs(X,Y);
161 for f being Function of X,Y st Y = {} implies X = {}
      holds TotFuncs <:f,X,Y:> = {f};
162 for f being Function of X,X holds TotFuncs <:f,X,X:> = {f};
164 for f being PartFunc of X,{y} for g being Function of X,{y}
      holds TotFuncs f = {g};
165 for f,g being PartFunc of X,Y
      st g c= f holds TotFuncs f c= TotFuncs g;
166 for f,g being PartFunc of X,Y
      st dom g c= dom f & TotFuncs f c= TotFuncs g holds g c= f;
167 for f,g being PartFunc of X,Y
      st TotFuncs f c= TotFuncs g & (for y holdsY <> {y})
      holds g c= f;
168 for f,g being PartFunc of X,Y
      st (for y holds Y <> {y}) & TotFuncs f =TotFuncs g holds f = g;
```

# B.12 SQUARE_1

Some Properties of Real Numbers Operations, by Andrzej Trybulec and Czeslaw Bylinski

```
reserve a,b,c,x,y,z for real number;
2 1 < x implies 1/x < 1;
5 2*a = a+ a;
6 a= (a-x)+x;
8 x - y=0 implies x = y;
ll x <  y implies 0 < y - x;
12 x <  y implies 0 <=y - x;
15 (x + x) /2 = x;
16 l/(l/x) =x;
17 x/(y*z) =x/y/z;
18 x*(y/z) = (x*y)/z;
19 0 <= x & 0 <= y implies 0 <= x*y;
20 x <= 0 & y <= 0 implies 0 <= x*y;
21 0 < x & 0 < y implies 0 < x*y;
22 x < 0 & y < 0 implies 0 < x*y;
23 0 <= x & y <= 0 implies x*y <= 0;
24 0 < x & y < 0 implies x*y < 0;
25 0 <= x*y implies 0 <= x & 0 <= y or x <= 0 & y <=0;
26 0 < x*y implies 0 < x & 0 < y or x < 0 & y <0;
27 0 <= a & 0 <= b implies 0 <= a/b;
29 0 < x implies y - x < y;

scheme RealContinuity { P[set], Q[set] } :
 ex z st
  for x,y st P[x] & Q[y] holds x <= z & z <= y
provided
 for x,y st P[x] & Q[y] holds x <= y;

def 1 func min(x,y) -> real number equals x if x <= y otherwise y;
def 2 func max(x,y) -> real number equals x if y <= x otherwise y;

34 min(x,y) = (x + y - abs(x - y)) / 2;
35 min(x,y) <= x;
38 min(x,y) = x or min(x,y) = y;
39 x <= y & x <= z iff x <= min(y,z);
40 min(x,min(y,z)) = min(min(x,y),z);
```

```
45 max(x,y) = (x + y + abs(x-y)) / 2;
46 x <= max(x,y);
49 max(x,y) = x or max(x,y) = y;
50 y <= x & z <= x iff max(y,z) <= x;
51 max(x,max(y,z)) = max(max(x,y),z);
53 min(x,y) + max(x,y) =x + y;
54 max(x,min(x,y)) = x;
55 min(x,max(x,y)) = x;
56 min(x,max(y,z)) = max(min(x,y),min(x,z));
57 max(x,min(y,z)) = min(max(x,y),max(x,z));

def 3 func x^2 equals x*x;

59 1^2 = 1;
60 0^2 = 0;
61 a^2 = (-a)^2;
62 (abs(a))^2 = a^2;
63 (a + b)^2 = a^2 + 2*a*b + b^2;
64 (a - b)^2 = a^2 - 2*a*b + b^2;
65 (a + 1)^2 = a^2 + 2*a + 1;
66 (a - 1)^2 = a^2 - 2*a + 1;
67 (a - b)*(a + b) = a^2 - b^2 & (a + b)*(a - b) = a^2 - b^2;
68 (a*b)^2 = a^2*b^2;
69 (a/b)^2 = a^2/b^2;
70 a^2-b^2 <> 0 implies 1/(a+b) = (a-b)/(a^2-b^2);
71 a^2-b^2 <> 0 implies 1/(a-b) = (a+b)/(a^2-b^2);
72 0 <= a^2;
73 a^2 = 0 implies a = 0;
74 0 <> a implies 0 < a^2;
75 0 < a & a < 1 implies a^2 < a;
76 1 < a implies a < a^2;
77 0 <= x & x <= y implies x^2 <= y^2;
78 0 <= x & x < y implies x^2 < y^2;

def 4 func sqrt a -> real number means 0 <= it & it^2 = a;

82 sqrt 0 = 0;
83 sqrt 1 = 1;
84 1 < sqrt 2;
85 sqrt 4 = 2;
86 sqrt 2 < 2;
89 0 <= a implies sqrt a^2 = a;
```

```
90 a <= 0 implies sqrt a^2 = -a;
91 sqrt a^2 = abs(a);
92 0 <= a & sqrt a = 0 implies a = 0;
93 0 < a implies 0 < sqrt a;
94 0 <= x & x <= y implies sqrt x <= sqrt y;
95 0 <= x & x < y implies sqrt x < sqrt y;
96 0 <= x & 0 <= y & sqrt x = sqrt y implies x = y;
97 0 <= a & 0 <= b implies sqrt (a*b) = sqrt a * sqrt b;
98 0 <= a*b implies sqrt (a*b) = sqrt abs(a)*sqrt abs(b);
99 0 <= a & 0 <= b implies sqrt (a/b) = sqrt a/sqrt b;
100 0 < a/b implies sqrt (a/b) = sqrt abs(a) / sqrt abs(b);
101 0 < a implies sqrt (1/a) = 1/sqrt a;
102 0 < a implies sqrt a/a = 1/sqrt a;
103 0 < a implies a /sqrt a = sqrt a;
104 0 <= a & 0 <= b
       implies (sqrt a - sqrt b)*(sqrt a + sqrt b)= a - b;
105 0 <= a & 0 <= b & a <>b
       implies 1/(sqrt a+sqrt b) = (sqrt a - sqrt b)/(a-b);
106 0 <= b & b < a
       implies 1/(sqrt a+sqrt b) = (sqrt a - sqrt b)/(a-b);
107 0 <= a & 0 <= b & a <> b
       implies 1/(sqrt a-sqrt b) = (sqrt a + sqrt b)/(a-b);
1O8 0 <= b & b < a
       implies 1/(sqrt a-sqrt b) = (sqrt a + sqrt b)/(a-b);
```

# B.13 REAL_2

Equalities and Inequalities in Real Numbers. Continuation of Real_1
byAndrzej Kondracki

```
reserve a,b,d,e for real number;

1 b-a=b implies a=0;
2 a+-b=0 or -a=-b or a-e=b-e or a-e=b+-e
      or e-a=e-b or e-a=e+-b implies a=b;
5 -a-b=-b-a;
6 -(a+b)=-a+-b & -(a+b)=-b-a;
8 -(a-b)=-a+b;
9 -(-a+b)=a-b & -(-a+b)=a+-b;
10 a+b=-(-a-b) & a+b=-(-a+-b) & a+b=a--b;
ll a=a+b+-b;
12 b=a-(a-b);
13 a+b=e+d implies a-e=d-b;
14 a-e=d-b implies a+b=e+d;
15 a-b=e-d implies a-e=b-d;
16 a+b=e-d implies a+d=e-b;
17 a=a+(b-b) & a=a+(b+-b) & a=a-(b-b) & a=a-(b+-b) & a=-b-(-a-b);
18 a-(b-e)=a+(e-b);
20 a+(-b-e)=a-b-e & a-(-b-e)=a+b+e;
22 a+b-e=a-e+b & a+b-e=-e+a+b;
23 a-b+e=e-b+a & a-b+e=-b+e+a;
24 a-b-e=a-e-b & a-b-e=-b-e+a & a-b-e=-e+a-b & a-b-e=-e-b+a;
25 -a+b-e=-e+b-a & -a+b-e=-e-a+b;
26 -a-b-e=-a-e-b & -a-b-e=-b-e-a & -a-b-e=-e-a-b & -a-b-e=-e-b-a;
27 -(a+b+e)=-a-b-e & -(a+b-e)=-a-b+e & -(a-b+e)=-a+b-e
     & -(-a+b+e)=a-b-e & -(a-b-e)=-a+b+e & -(-a+b-e)=a-b+e
     & -(-a-b+e)=a+b-e & -(-a-b-e)=a+b+e;
28 a+e=(a+b)+(e-b) & a+e=(a+b)-(b-e);
29 a-e=(a-b)-(e-b) & a-e=(a-b)+(b-e) & a-e=(a+b)-(e+b);
30 b<>0 implies (a/b=1 or a*b"=1 implies a=b);
31 e<>0 & a/e=b/e implies a=b;
33 a"=b" or 1/a=1/b or 1/a=b" implies a=b;
34 b<>0 & a/b=-1 implies a=-b & b=-a;
35 a*b=l implies a=l/b & a=b"
36 b<>0 implies (a=l/b or a=b" implies a*b=l & a"=b& b=l/a);
```

```
37 b<>0 & a*b=b implies a=1;
38 b<>0 & a*b=-b implies a=-1;
39 a<>0 & b<>0 & b/a=b implies a=1;
40 a<>0 & b<>0 & b/a=-b implies a=-1;
41 a<>0 implies 1/a<>0;
42 a<>0 & b<>0 implies a*b"<>0 & a/b<>0& a"*b"<>0 & 1/(a*b)<>0;
45 (-a)"=-a" & (a<>0 implies (-a)/a=-1 & a/(-a)=-1);
46 a<>0 implies (a=a" or a=1/a implies a=1 or a=-1);
47 (a*b")"=a"*b & (a"*b")"=a*b;
48 1/(a/b)=b/a & (a/b)"=b/a;
49 (-a)*(-b)=a*b & -a*(-b)=a*b & -(-a)*b=a*b;
50 b<>0 implies (a/b=0 iff a=0);
51 (1/a)*(1/b)=1/(a*b);
53 (a/e)*(b/d)=(a/d)*(b/e);
55 e<>0 implies a/b=(a/e)/(b/e)
      & a/b=a/(b*e)*e & a/b=e*(a/e/b) & a/b=a/e*(e/b);
56 a*(1/b)=a/b;
57 a/(1/b)=a*b;
58 -a/(-b)=a/b & -(-a)/b =a/b & (-a)/(-b)=a/b & (-a)/b=a/(-b);
61 a/(b/e)=a*(e/b) & a/(b/e)=e/b*a & a/(b/e)=e*(a/b) & a/(b/e)=a/b*e;
62 b<>0 implies a=a*(b/b) & a=a*b/b & a=a*b*(1/b)
      & a=a/(b/b) & a=a/(b*(1/b)) & a=a*(1/b*b)& a=a*(1/b)*b;
63 for a,b ex e st a=b-e;
64 for a,b st a<>0 & b <>0 ex e st a=b/e;
65 b<>O implies a/b+e=(a+b*e)/b;
66 b<>0 implies a/b-e=(a-e*b)/b & e-a/b=(e*b-a)/b;
67 a/b/e=a/e/b & a/b/e=1/b*(a/e) & a/b/e=1/e*(a/b) & 1/e*(a/b)=a/(b*e);
70 (a*b)/(e*d)=(a/e*b)/d;
71 (-1)*a=-a & (-a)*(-l)=a & -a=a/(-l) & a=(-a)/(-l);
74 a<>0 & e<>0 & a=b/e implies e=b/a;
75 e<>0 & d<>0 & a*e=b*d implies a/d=b/e;
76 e<>0 & d<>0 & a/d=b/e implies a*e=b*d;
77 e<>0 & d<>0 & a*e=b/d implies a*d=b/e;
78 b<>0 implies a*e=a*b (e/b);
79 b<>0 & e<>0 implies a*e=a*b/(b/e);
80 a/b*e=e/b*a & a/b*e=1/b*a*e & a/b*e=1/b*e*a;
82 b<>0 & d<>0 & b<>d & a/b=e/d implies a/b=(a-e)/(b-d);
83 b<>0 & d<>0 & b<>-d & a/b=e/d impliesa/b=(a+e)/(b+d);
85 (a-b)*e=(b-a)*(-e) & (a-b)*e=-(b-a)*e;
88 3*a=a+a+a & 4*a=a+a+a+a;
89 (a+a+a)/3=a & (a+a+a+a)/4=a & (a+a)/4=a/2;
90 a/4+a/4+a/4+a/4=a;
```

```
91 a/(2*b)+a/(2*b)=a/b & a/(3*b)+a/(3*b)+a/(3*b)=a/b;
92 e<>0 implies a+b=e*(a/e+b/e);
93 e<>0 implies a-b=e*(a/e-b/e);
94 e<>0 implies a+b=(a*e+b*e)/e;
95 e<>0 implies a-b=(a*e-b*e)/e;
96 a<>0 implies a+b=a*(1+b/a);
97 a<>0 implies a-b=a*(1-b/a);
98 (a-b)*(e-d)=(b-a)*(d-e);
99 (a+b+e)*d=a*d+b*d+e*d & (a+b-e)*d=a*d+b*d-e*d
      & (a-b+e)*d=a*d-b*d+e*d & (a-b-e)*d=a*d-b*d-e*d;
100 (a+b+e)/d=a/d+b/d+e/d & (a+b-e)/d=a/d+b/d-e/d
      & (a-b+e)/d=a/d-b/d+e/d & (a-b-e)/d=a/d-b/d-e/d;
101 (a+b)*(e+d)=a*e+a*d+b*e+b*d & (a+b)*(e-d)=a*e-a*d+b*e-b*d
      & (a-b)*(e+d)=a*e+a*d-b*e-b*d & (a-b)*(e-d)=a*e-a*d-b*e+b*d;
105 (a+-b<=0 or b-a>=0 or b+-a>=0
      or a-e<=b+-e or a+-e<=b-e or e-a>=e-b) impliesa<=b;
106 (a+-b<0 or b-a>0 or -a+b>0
      or a-e<b+-e or a+-e<b-e or e-a>e-b) impliesa<b;
109 a<=-b implies a+b<=0 & -a>=b;
110 a<-b implies a+b<0 & -a>b;
111 -a<=b implies a+b>=0;
112 -b<a implies a+b>0;
117 b>0 implies (a/b>1 implies a>b) & (a/b<1 implies a<b)
    & (a/b>-1 implies a>-b & b>-a) & (a/b<-1implies a<-b & b<-a);
118 b>0 implies (a/b>=1 implies a>=b) & (a/b<=1 impliesa<=b)
    & (a/b>=-1 implies a>=-b & b>=-a) &(a/b<=-1 implies a<=-b & b<=-a);
119 b<0 implies (a/b>1 implies a<b) & (a/b<1 implies a>b)
    & (a/b>-1 implies a<-b & b<-a) & (a/b<-1implies a>-b & b>-a);
120 b<0 implies (a/b>=1 implies a<=b) & (a/b<=1 impliesa>=b)
    & (a/b>=-1 implies a<=-b & b<=-a) &(a/b<=-1 implies a>=-b & b>=-a);
121 a>=0 & b>=0 or a<=0 & b<=0 implies a*b>=0;
122 a<0 & b<0 or a>0 & b>0 implies a*b>0;
123 a>=0 & b<=0 or a<=0 & b>=0 implies a*b<=0;
125 a<=0 & b<0 or a>=0 & b>0 implies a/b>=0;
126 a>=0 & b<0 or a<=0 & b>0 implies a/b<=0;
127 a>0 & b>0 or a<0 & b<0 implies a/b>0;
128 a<0 & b>0 implies a/b<0 & b/a<0;
129 a*b<=0 implies a>=0 & b<=0 or a<=0 & b>=0;
132 a*b<0 implies a>0 & b<0 or a<0 & b>0;
133 b<>0 & a/b<=0 implies b>0 & a<=0 or b<0 &a>=0;
134 b<>0 & a/b>=0 implies b>0 & a>=0 or b<0 &a<=0;
135 b<>0 & a/b<0 implies b<0 & a>0 or b>0 &a<0;
```

```
136 b<>0 & a/b>0 implies b>0 & a>0 or b<0 &a<0;
137 a>=1 & b>=1 or a<=-1 & b<=-1 implies a*b>=1;
138 a>=1 & b>=1 or a<=-1 & b<=-1 implies a*b>=1;
139 0<=a & a<1 & 0<=b & b<=1 or 0>=a & a>-1& 0>=b & b>=-1
      implies a*b<1;
140 0<=a & a<=1 & 0<=b & b<=1 or 0>=a & a>=-1 & 0>=b & b>=-1
     implies a*b<=1;
142 0<a & a<b or b<a & a<0 implies a/b<1 & b/a>1;
143 0<a & a<=b or b<=a & a<0 implies a/b<=1 &b/a>=1;
144 a>0 & b>1 or a<0 & b<1 implies a*b>a;
145 a>0 & b<1 or a<0 & b>1 implies a*b<a;
146 a>=0 & b>=1 or a<=0 & b<=1 implies a*b>=a;
147 a>=0 & b<=1 or a<=0 & b>=1 implies a*b<=a;
149 a<0 implies 1/a<0 & a"<0;
150 (1/a<0 implies a<0) & (1/a>0 implies a>0);
151 (0<a or b<0) & a<b implies 1/a>1/b;
152 (0<a or b<0) & a<=b implies 1/a>=1/b;
153 a<0 & b>0 implies 1/a<1/b;
154 (1/b>0 or 1/a<0) & 1/a>1/b implies a<b;
155 (1/b>0 or 1/a<0) & 1/a>=1/b implies a<=b;
156 1/a<0 & 1/b>0 implies a<b;
157 a<-1 implies 1/a>-1;
158 a<=-1 implies 1/a>=-1;
164 1<=a implies 1/a<=1;
165 (b<=e-a implies a<=e-b) & (b>=e-a implies a>=e-b);
167 a+b<e+d implies a-e<d-b;
168 a+b<e+d implies a-e<d-b;
169 a-b<e-d implies a+d<e+b & a-e<b-d & e-a<d-b &b-a<d-e;
170 a-b<e-d implies a+d<e+b & a-e<b-d & e-a<d-b &b-a<d-e;
171 (a+b<e-d implies a+d<e-b) & (a+b<e-d implies a+d<e-b);
173 (a<0 implies a+b<b & b-a<b) & (a+b<b or b-a<bimplies a<0);
174 (a<=0 implies a+b<=b & b-a>=b) & (a+b<=b or b-a>=b implies a<=0);
177 (b>0 & a*b<=e implies a<=e/b) & (b<0 & a*b<=eimplies a>=e/b) &
     (b>0 & a*b>=e implies a>=e/b) & (b<0& a*b>=e implies a<=e/b);
178 (b>0 & a*b<e implies a<e/b) & (b<0 & a*b<eimplies a>e/b) &
     (b>0 & a*b>e implies a>e/b) & (b<0& a*b>e implies a<e/b);
181 (for a st a>0 holds b+a>=e)
        or (for a st a<0 holds b-a>=e) impliesb>=e;
182 (for a st a>0 holds b-a<=e)
        or (for a st a<0 holds b+a<=e) impliesb<=e;
183 (for a st a>1 holds b*a>=e)
        or (for a st 0<a & a<1 holds b/a>=e)implies b>=e;
```

```
184 (for a st 0<a & a<1 holds b*a<=e)
        or (for a st a>1 holds b/a<=e) impliesb<=e;
185 (b>0 & d>0 or b<0 & d<0) & a*d<e*b impliesa/b<e/d;
186 (b>0 & d<0 or b<0 & d>0) & a*d<e*b impliesa/b>e/d;
187 (b>0 & d>0 or b<0 & d<0) & a*d<=e*b impliesa/b<=e/d;
188 (b>0 & d<0 or b<0 & d>0) & a*d<=e*b impliesa/b>=e/d;
193 b<0 & d<0 or b>0 & d>0 implies
      (a*b<e/d implies a*d<e/b) & (a*b>e/dimplies a*d>e/b);
194 b<0 & d>0 or b>0 & d<0 implies
      (a*b<e/d implies a*d>e/b) & (a*b>e/dimplies a*d<e/b);
197 (0<a or 0<=a) & (a<b or a<=b) & (0<e or 0<=e)& e<=d
       implies a*e<=b*d;
198 0>=a & a>=b & 0>=e & e>=d implies a*e<=b*d;
199 0<a & a<=b & 0<e & e<d or 0>a & a>=b& 0>e & e>d implies a*e<b*d;
200 (e>0 & a>0 & a<b implies e/a>e/b) &
     (e>0 & b<0 & a<b implies e/a>e/b);
201 e>=0 & (a>0 or b<0) & a<=b implies e/a>=e/b;
202 e<0 & (a>0 or b<0) & a<b implies e/a<e/b;
203 e<=0 & (a>0 or b<0) & a<=b implies e/a<=e/b;
204 for X,Y being Subset of REAL st
      X<>{} & Y<>{} & for a,b st a inX & b in Y holds a<=b
      holds ex d st (for a st a in X holds a<=d) &for b st b in Y
      holds d<=b;
```

# Appendix C

## C.1 Contacting the Mizar Society

### C.1.1 Mizar Society

Association of Mizar Users
University of Bialystok
Institute of Mathematics
ul. Akademicka 2
15-267 Bialystok, Poland

Fax: +48-85-745-75-45
E-mail: mus@mizar.uwb.edu.pl
WWW: http://mizar.org/

### C.1.2 Mizar Society Nagano Circle

4-17-1 Wakasato Nagano-shi, Nagano-ken, 380-8553 JAPAN

Shinshu University, Information Engineering of the Engineering Faculty
Nakamura lab.
Mizar Society Nagano Circle

Fax: +81-26- 269-5495
E-mail: kiso@cs.shinshu-u.ac.jp

## C.2 Acquisition of Mizar by ftp

Mizar can be acquired by anonymous ftp from the ftp server of Japan and Poland which are shown below.

Japan:

`ftp://markun.cs.shinshu-u.ac.jp/pub/mizar/`

`ftp://nicosia.is.s.u-tokyo.ac.jp/pub/misc/pcmizar/`

Poland:

`ftp://ftp.mizar.org/`

`ftp://mizar.uwb.edu.pl/pub/system/`

`ftp://sunsite.icm.edu.pl/pub/mizar/`

The file names of Mizar are the following, `X` shows the version of Mizar software, and `Y` shows the version of MML.

| | |
|---|---|
| MS-DOS version: | `mizar-X_Y-win32.exe` |
| Linux version: | `mizar-X_Y-linux.tar` |

For example, when `X` is `6.1.12` and `Y` is `3.33.722`, "`mizar-6.1.12_3.33.722-win32.exe`" is the MS-DOS version Mizar file of which Mizar Version is 6.1.12 and MML vsersion is 3.33.722. In addition, the file of a MS-DOS version is a self-extracting file.

## C.3 How to submit Articles via the Internet

We attach the accomplished MIZ file, the VOC file, the BIB file, etc. on e-mail, and send them. If they are attached with a text, fault may arise in conversion of `8bit->7bit`.

So, for example, it is possible to send, after specifying a type and encoding as follows.

```
    Type: audio/x-mpeg
    Encoding: base64
```

Or it is also good to attach them to e-mail and to send them, after compressing them into the file of general compressed format (for example, PKZIP, ARJ, RAR and TAR, GZIP, etc.).

A destination is the MIZAR Library Committee:

```
    mml@mizar.uwb.edu.pl
```

## C.4 WWW Homepage

The URL for the Mizar homepage is `http://mizar.org/`.

Here, all abstract files are put up in html format. Moreover, since a link can also be followed about where terminology was defined first, it is very convenient.

And then, these mirror servers are in Shinshu University of Japan, University of Alberta of Canada, etc.

Mirror servers:
`http://mizar.uwb.edu.pl/`
(University of Bialystok, Bialystok, Poland)

`http://www.cs.ualberta.ca/~piotr/Mizar/mirror/http/`
(University of Alberta, Edmonton, Canada)

http://markun.cs.shinshu-u.ac.jp/Mirror/mizar/htdocs/
(Shinshu University, Nagano, Japan)

## C.5 Formalized Mathematics

When the Mizar articles are accepted, they are automatically translated into English theses and are placed in Formalized Mathematics. This is published from the Warsaw University Bialystok Branch several times per year. Up until the end of 1998, a total of 7 volumes have been published, all articles registered into the Mizar library are entered in these volumes as an English paper.

For more information about Formalized Mathematics, please contact the following.

Fondation for Information Technology
Logic and Mathematics
Krochmalna 3/917
00-864 Warsaw
Poland

Fax: +48(85) 745.74.78
E-mail: romat@mizar.org

Formal name of Formalized Mathematics:

Formalized Mathematics,
Edited by Warsaw University-Bialystok Branch,
Roman Matuszewaki.
ISSN 1426-2630.

# Conclusion

This original book was improved and edited by Watanabe and Tanaka based on the lectures by Nakamura on Sep.5th, 1992 at Shinshu University.

Mizar of those days was based on the version 3.29, and some part of it was based on the Mizar version 4.09.

The contents of this book are the 4th edition based on the version 6.1.12 which is the latest version as of June, 2002 through the revised edition which is referring to the Mizar version 5.2.12, and the 3rd edition which is referring to the version 5.3.06.

Please note that Mizar versions are frequently updated.

Authors

# Reference

[1] Ewa Bonarska (Olgierd Wojtasiewicz English Translation, Roman Matuszewski Edition), An Introduction to PC Mizar, Fondation Philippe le Hodey, 1990

# Index