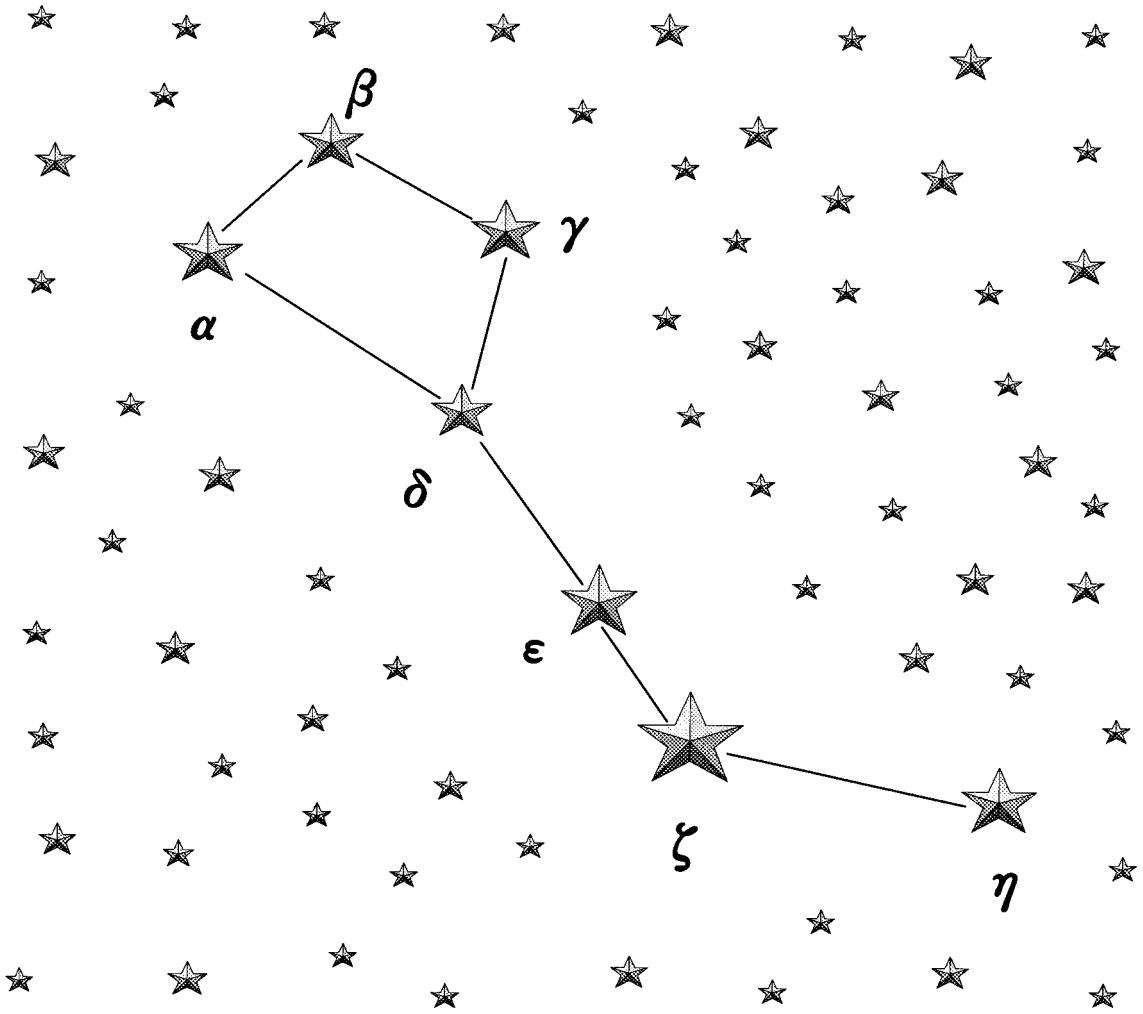
Mizar 講義録

（四訂版：Mizar Version 6.1.12）

中村八束，渡辺稔彦，
田中保史，カワモト・ポーリン　共著



信州大学工学部　情報基礎研究室

目次

まえがき 1

改訂版にあたって 2

1 Mizar の概要 3

2 Mizar 言語の概要 5

3 Mizar article の構成 9

3.1 全体構成 ····· 9

3.2 環境部 ····· 10

3.2.1 vocabulary 部 ····· 10

3.2.2 notation 部，constructors 部 ····· 11

3.2.3 requirements 部 ····· 13

3.2.4 definitions 部 ····· 14

3.2.5 theorems 部 ····· 14

3.2.6 schemes 部 ····· 14

3.3 本体部 ····· 15

3.3.1 文 ····· 15

3.3.2 定理 ····· 16

4 証明の方法 19

4.1 基本的な証明方法 ····· 19

4.2 背理法 ····· 21

4.3 証明の分割 ····· 22

4.4 同値の証明 ····· 23

4.5 now の使い方 ····· 23

4.5.1 end についての注意 ····· 24

4.6 全称記号 ····· 24

4.6.1 変数が 2 つある場合 ····· 25

4.6.2 such that を使った書き方 ····· 26

4.6.3 文の分割 ····· 26

4.6.4 文の分割 assume での場合 27
4.6.5 that の後の注意 28
4.7 存在記号 29
4.7.1 consider の使い方 29
4.7.2 given の使い方 30
4.8 場合分け 31
4.8.1 場合分けの前提 32
4.8.2 場合わけでのラベル 32
4.8.3 ラベルについての注意 33
4.9 @proof の使い方 33
4.10 新しい変数の導入 (Set, reconsider) 34
4.11 take について 34
4.12 hereby 36

5 mode について 39
5.1 mode 39
5.2 階層構造 40
5.3 mode の変更 40
5.4 証明中の mode の宣言 41
5.4.1 c= についての注意 41
5.4.2 実際に definition を使っている例 42

6 定義 43
6.1 定義 43
6.2 vocabulary file の書き方 43
6.3 predicate の定義 44
6.4 functor の定義 45
6.5 equqls の使い方 47
6.6 mode の定義 48
6.7 Cluster の定義 50
6.8 structure の導入 52

7 実行環境 53
7.1 インストール 53
7.1.1 MS-DOS 版 54
7.1.2 Linux 版 55
7.2 Mizar を使うための準備 57

7.3 Mizar のシステム ····· 58
7.4 Mizar の使い方 ····· 59
7.4.1 accommodate ····· 59
7.4.2 Mizar チェッカー ····· 59
7.4.3 エラーがある場合 ····· 60
7.4.4 便利なコマンド ····· 60

8 ライブラリへの登録 61
8.1 article の改良 ····· 61
8.1.1 本体部の改良 ····· 61
8.1.2 環境部のチェック ····· 63
8.2 登録の準備 ····· 64

9 便利なツール 67
9.1 用語のある VOC ファイルを探す (findvoc) ····· 67
9.2 VOC ファイルの中味を知る ····· 68
9.3 その用語が使用されている ABS ファイルを知る ····· 68
9.3.1 grep を使う方法 ····· 68
9.3.2 Web から検索する方法 ····· 69

10 バージョンアップについて 71

A 予約語一覧 73

B Mizar ライブラリ定理集 75
B.1 TARSKI ····· 76
B.2 AXIOMS ····· 77
B.3 BOOLE ····· 78
B.4 XBOOL_0 ····· 79
B.5 XBOOL_1 ····· 81
B.6 REAL_1 ····· 84
B.7 NAT_1 ····· 87
B.8 FUNCT_1 ····· 90
B.9 SUBSET_1 ····· 95
B.10 FINSEQ_1 ····· 98
B.11 FUNCT_2 ····· 103
B.12 SQUARE_1 ····· 111
B.13 REAL_2 ····· 114

C C.1 Mizar Society 連絡先 **119**
C.1.1 Mizar Society ····· 119
C.1.2 Mizar Society Nagano Circle ····· 119
C.2 ftp による Mizar の入手法 ····· 120
C.3 インターネットの投稿法 ····· 120
C.4 WWW Homepage ····· 121
C.5 Formalized Mathematics ····· 122

あとがき **123**
参考文献 **123**
索引 **124**

まえがき

Mizar は，もっとも有名なプルーフチェッカーの 1 つです．歴史的にはかなり古い方ですし，それから，本格的な数学用言語としてはいちばん完成度が高いものだということがいえると思います．プルーフチェッカー自体は世界中に何十種類とあるようですが，それらは大学院の学生が卒論でつくって，それでその学生が卒業したら誰も使わなくなってしまったようなものがたくさんあるわけです．その中にあって，Mizar はワルシャワ大学の Andrzej Trybulec 先生を中心とした Mizar グループによって，多くの研究者や大学院生などがそれに協力してライブラリを増やしています．

Trybulec 先生はもともとは数学者であるため，どんなプルーフチェッカーが必要であるかということをよく知っておられます．したがって，そういう経験をもとに作られましたので機能も充実していると思います．このプルーフチェッカーというものは，昔にはもちろんこういうものはなくて数学は非常に厳密な学問であると思われていました．多くの人が機械に頼るまでもないと思っていました．しかし，コンピュータの発達に伴って，その厳密さというものに疑問が感じられるようになってきました．そして，その数学をまずどういうふうに記述すべきかということが問題になったわけです．プルーフチェッカーは皆それぞれ考え方が違っています．私が作った THEAX もそのうちの 1 つですけれども， Mizar とはまったく独立に，時期的にはほとんど同じか，少し THEAX の方が遅れてるかもしれませんが，ほぼ同じ頃に独立につくっています．したがって考え方がまったく違っています．

ここでは Mizar の考え方，どういう考え方で数学を形式化したかということの話，そしてまた実際の動作環境についても説明していきたいと思います．

改訂版にあたって

今日の Mizar は，ますます多くの応用分野で使われるようになり，インターネットの WWW ホームページも持つようになりました．ユーザ達のサポートや Mizar の処理能力を上げるために，Mizar は頻繁にバージョンアップをしています．この講義録に，その改良をできるだけまとめていき，これからも Mizar を学ぶ人の助けになることを期待しています.

四訂版にあたって

本書の内容は Mizar バージョン 6.1.12 に基づきます.

Chapter 1

Mizar の概要

Mizar とは，ワルシャワ大学(ポーランド)の Andrzej Trybulec 先生を中心とした Mizar Society によってすすめられている，数学をコンピュータを使って形式化するプロジェクトの総称です． Mizar プロジェクトは，コンピュータを使って数学を記述するために作られた Mizar 言語によって数学の証明を記述し，それを IBM-PC 上で動作する Mizar プルーフチェッカーによってチェックし，それを Mizar ライブラリに登録することによって行われます．このプロジェクトの目的は数学の論文のチェックのシステムを作ることです． Mizar では，数学の証明を記述したテキストのことを article といいます．新たに article を書くときには，すでにチェックされていて Mizar ライブラリに登録されている多くの article を参照することができます．そして，その article がいずれ Mizar ライブラリに登録されたときには，それを他の article が参照することもできます.

article は，環境部 (environ)，本体部 (text proper) に分かれており，環境部では，その article に必要な様々な環境設定を記述し，本体部では証明自体を記述します.

article は Mizar 言語によって記述します． Mizar 言語は，数学の一般的な証明の記述のしかたをもとにしていますが，Mizar 独特の書き方もあり，これについては後で詳しく説明します.

ここでは Mizar 言語，Mizar プルーフチェッカー，さらに Mizar プロジェクトについて説明します.

Chapter 2
Mizar 言語の概要

Mizar 言語は，コンピュータを使って数学の証明を記述するための言語です．普段，証明を書くときに証明の書き方にしたがって書くように，Mizar article を書くときには Mizar 言語で書かなければなりません．以下では Mizar 言語を Mizar と略します．

Mizar では一般的なアスキーコードを使用します．通常使用する文字は，数字，英字，記号です．また，数学の記号を一般的なアスキーキャラクタを組みあわせたり，似たものを使うものもあります．

また， Mizar では， 英字の大文字と小文字は区別されることに注意してください．但し，article 名など，MS-DOS のファイル名になっているものを article 中に記述する場合，必ず大文字で書きます．この場合 8 文字までの英大文字，数字，__，' で書きます．

数学の証明を書くときに，'したがって'， '定理'， '定義'， '証明終り'などの言葉を使うように，Mizar でもそれに対応する言葉を使って記述します．これらの言葉は予約語になっています．Mizar のシステムとして， `&`， `or`， `not` の論理， = の概念，All, Exist の述語論理は最初から備わっています．さらに，Mizar では自分の論文 (article) に，Mizar ライブラリから自由に functor (関数記号) ， predicate (述語論理)などの定義， 定理を引用し証明を進めていきますが，その基本となるものが HIDDEN.ABS に書かれています．ここに書かれていることは，それを公理として証明なしで認めています．それを引用してさまざまな定理が導かれていて Mizar ライブラリを構成しています．ユーザが新たに article を書く場合，最初からシステムに備わっている論理や，TARSKI などをもとにして出来ているたくさんの定理を引用して記述していきます．

ユーザが新たに functor 函数などを定義する場合，その functor などを vocabulary file に登録し，article の中で functor を定義します．この場合，予約語とまったく同じ名前の

functor は定義することはできません．具体的な定義のしかたについては第 6 章定義で説明します．また， 付録に Mizar 予約語一覧表がありますので参照してください．

Mizar のシステムに備わっている論理式と述語論理について説明します．Mizar では基本的な論理式，&, or, not, implies, iff を使うことができます例えば，&

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

と書くことができます．& は論理式の中の場合 & を使い，式と式との関係の場合 and を使いますまた， implies は ’ならば’， iff は if and only if ’同値’，を表します．述語論理ですが，次のように全称記号と存在記号を使うことができます．

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

これは， 通常の述語論理の記法では次の式を表します．

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

これは， 任意の x について， $f(x) = b$ ならば $g(x) = c$ という意味です．$f(x) = b$ を満たすような， すべての x について $g(x) = c$ が成立するということです．

なお， f[x]， g[x] とは x を含んだ述語ということで，こういう記法がシステムに備わっているわけではありません．

次に，

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

と書くとこれは，

$$(\exists x)(\mathrm{f}\ (\mathrm{x}) = \mathrm{b}\ \text{and}\ \mathrm{g}\ (\mathrm{x}) = \mathrm{c}\ )$$

を意味します．つまり $f(x) = b$ かつ $g(x) = c$ となる x が存在するという意味です．また， 全称記号と存在記号を混在して使うこともできます．

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

これは,

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

を表します.

括弧は任意に使うことを許されています．たとえ括弧が必要ないときでも，わかりやすくするために括弧を使うことができます.

なお，Mizar のシステムには，MS-DOS 版と Linux 版があります．さらに本質的には Linux 版なのですが，Web ページから自由に使えるものもあります.

Chapter 3
Mizar article の構成

3.1 全体構成

次に article の構成について説明します．article は，環境部，本体部の 2 つの部分からなります．環境部は environ で始まり，セミコロンで終わります．本体部は begin で始まり，最後はセミコロンです．但し，本 begin をいくつか入れて本体部をいくつかに分けることもできます．これは論文を節に分けることに相当します．後に述べる .BIB ファイルに，節の名前を書いておくと，後に通常の数学論文の形式に自動変換されたときに， begin の位置に節の名前が書き込まれます.

```
environ
          環境部
                    ;
 begin
         本体部 1
                    ;
 begin
         本体部 2
 ― ― ―
 begin
         本体部 n
                    ;
```

3.2 環境部

まず環境部について説明します．環境部には，その article で必要な環境設定を行います．具体的にはその article が引用する定義，定理，または vocabulary がある article のファイル名を書き宣言します.

環境部は次のような項目からなっています.

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

環境部の中のそれぞれの項目について説明します.

3.2.1 vocabulary 部

vocabulary 部には，article の中で使われている vocabulary の登録されている vocabularyfile (.VOC ファイル)のファイル名を大文字で書きます． ファイル名の拡張子は省略し，複数の vocabulary file を書く場合はコンマ(,)で区切って書きます[1]． 行末にはセミコロン(；)を書きます．また，vocabulary 部以外もこれと同様に書きます.

Mizar ライブラリに登録されている article で使用されている vocabulary が登録されている vocabulary file は， 1 つの特殊な形式のファイルにまとめられています.

使用する vocabulary が登録されている vocabulary file を探すには，MS-DOS のコマンドプロンプト,または Linux のプロンプトで

```
FINDVOC [vocabulary]
```

とします．また，vocabulary file に登録されている vocabulary を表示するには,

```
LISTVOC [vocabulary]
```

とします.

---

1 HIDDEN.VOC に登録されている vocabulary については,vocabulary 部に書かなくても使うことができます.

3.2.2 notation 部. constructors 部

新しい理論を作る際, 古い(ライブラリ中の)概念を使いたいときがあります. 例えば Function とか Function-like という概念を使いたいとします. このとき, これらの用語は FUNC.VOC という vocabulary ファイルに登録されていますので, vocabulary 部に FUNC を加えます. また, これらの用語は FUNCT_1 の中で(MIZ 又は ABS ファイル)定義されていますので, notation 部に FUNCT-1 を加えます. そして, 多くの場合 constructors 部にも FUNCT_1 を加えます.

例

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

notation 部と constructors 部の間の違いですが, 昔の MIZAR のバージョンでは signature 部というのが 1 つだけあって, 両者は統合されていました. そこから概念間の階層構造等の関係と個々の概念の定義が, コンピュータのメモリに展開されました. しかし前者は重複する部分が多かった為, メモリの節約のために分離されたのです. 前者は constructors 部に入れることになりました. そのため, より包含的な構造をもつファ

イルがあれば，より小さい構造のファイルは constructors 部から省略でき，メモリの節約ができるようになったのです．例えば，上の例のかわりに，

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

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

としてもエラーは出ません，従って TOPREAL1 が constructors 部にあらかじめ入っている場合には，そこに FUNCT_1 は入れなくてよいのです．

最初は notation 部と constructors 部両方にファイル名を順に入れてゆき，完成後，よりベーシックなファイルを constructors 部から削ってみると環境部をより単純にすることができ，またメモリも節約できます．

notation 部には，vocabulary ファイル中に書いたシンボルを使用したい場合，そのシンボルが定義されている article のファイル名をここに書いておきます．article の定義部 (definition の部分)では，そのシンボルの意味付けがなされています(どういうアーギュメントをいくつ持つのか，そのシンボル自身がどういうモードになるのか等)．

例えば，article BOOLE と PRE_TOPC にはどちらにも∩ (インターセクション)が定義されています(後者は再定義)．しかし，そのシンボルの使われ方が異なっているのです．

< article BOOLE >

```
definition
  let X,Y;
  func X /\ Y -> set means
  x c= it iff x in X & x in Y;
end;
```

< article PRE_TOPC >

```
definition
  let TP, P, Q;
  redefine
  func P /\ Q -> Subset of TP;
end;
```

BOOLE での /\ (∩のこと）は，それ自身 set を意味し， アーギュメントとして set を両側に持ちます.

それに対して PRE_TOPC でのそれは，自分自身を TP(トポロジカルスペース)の Subset とし， アーギュメントとして TP の Subset を両側に持ちます.

/\を使った article をこれから書こうとする場合， それを set として使用したければ notation 部に BOOLE と書き， TP の Subset としたければ PRE_TOPC と書かなければいけません.

3.2.3 requirements 部

requirements 部は今のところ ARYTM というファイルだけを置きます．ARYTM は算術計算を比較的自由に使えるようにするためのファイルです.

例えば,

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

というような式は理由 (by....)をつけずに真となりますが，これは ARYTM お陰です (但し 132 - 24 = 108 には理由が必要).

例えば,

```
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;
```

はノーエラーです．乗算，不等式も自動的に判断されます．ARYTM は notation にも入れられていることに注意しましょう．

3.2.4 definitions 部

definitions 部では，証明をする際の定義の拡張，証明の飛躍 (definitional expansion) を可能にしたい場合に，その定義がなされている article のファイル名を書きます．例えば，x c= y を証明するとき，∀a(a in x implies a in Y) を証明すればよいのですが，この定義は article TARSKI の中に以下のように定義されています．

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

よって，definitions 部に TARSKI と書いておけば x c= y を証明するとき，
a in x implies a in Y をいってしまえば x c= y が証明されたことになります．

例，

```
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 部

theorems 部では，article に引用する定理，定義が書かれている article のファイル名を書きます.

3.2.6 schemes 部

schemes 部では，引用するスキーム (証明の型) の書かれている article のファイル名を書きます.

3.3 本体部

3.3.1 文

次に本体部の説明をします．本体部は，基本的には文からなり，文の並んだものです. 文は次のような構造をしています.

```
   L1: A=B      by ...;
   L2: B=C      by ...;
   L3: A=C      by L1,L2;
ラベル　式       引用部
```

ラベルは必要な文(後で引用する文)だけつけます．また，by ...の部分を引用部といい，その式を導くのに必要な文のラベルを書きます．引用部， by ...では，その article 中のラベルと他の article からの引用のラベルを混ぜることができます．例えば,

```
   ..... by L1,L2,TARSKI:5;
```

というふうに,その article 中のラベルはそのまま，他の article からの引用の場合は, article 名 : 数字です．また定義を引用する場合，article 名 :def　数字です.

さらに，引用部の代わりに,

```
L3:  A=C
proof
 ...
```

```
 ...
 hence thesis;
end;
```

というふうに，proof 〜 end ではさんだいくつかの文で証明をつけることもできます．この場合，これ全体で 1 つの大きな文とみなせます．このとき，proof 〜 end の end の直前には thus もしくは hence という文を必要とします．これは，いま証明を付けている最初の式がこの文の式によっていえています，ということです．thus と hence の違いですが，hence は then + thus になっています．

then はここではじめて出てきましたが，直前の式を引用するときに使います．例で説明すると，

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

となっているとします. このときに,

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

と書く代わりに，L2 の式は直前の式ですから，L3 の文の引用部から L2 を取り除いて，then を使って,

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

と書くことができます.

then を使う理由には，ラベルの数をなるべく減らそうということがあります．ふつう証明では直前の式を引用することが非常に多くあります．直前の式を引用するときにも，ラベルをつけて，by ラベルとすると，たくさんのラベルを必要とします．ですから，直前の式を引用するときはラベルを使って書く書きかたも間違いではありませんが，then と書いた方が良いわけです.

また，article を Mizar ライブラリに登録するときに使う article の改良のためのチェッカー(Chapter7 参照) でチェックするときに，証明は正しくても上記のラベルを使って書く方式は，皆エラーになってしまいます．そのエラーがある限り，Mizar ライブラリに

は登録させてもらえませんから，結局のところ書いてはいけないということです．
then が使えるときにはラベルを使わないで then を使います．このため，proof ～ end の end の直前の文で，その直前の式を引用するときは hence を使います．

3.3.2 定理

一般に本体部は定理と定義を中心にして構成されています．
article で，定理を書くときには，

```
theorem 定理の式
proof
  ...
  ...
end;
```

と，書きます．
こう書くのと先ほどの，

```
   式
proof
 ...
 ...
end;
```

と，書くのではどう違うかというと，theorem と書くことによって，その式が abstract file に定理として抜きだされるということです．abstract file とは，article から theorem (定理の式) や definition (定義の式) などを抜きだしたファイルです．(.ABS ファイル) article を，Mizar ライブラリに登録するときには，article を abstract file にして登録します．なぜなら，証明などをすべて含んだ article (.MIZ ファイル)では，ファイルの大きさが膨大になってしまって，そこから引用するものを見つけるのは大変だからです．ですから，引用してもらいたい式を書くときには theorem をつけておきま

す.

article で定義を書くときには definition を使います．なにか新しい語を定義するとか，新しい関数を定義するときは，この definition を使って定義します．定義については後で詳しく説明します.

それから，theorem や definition は proof 〜 end の中に書くことはできません．ネスティングの一番上の段階だけで使うことができます．つまり，

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

と書くことはできません.

このように，Mizar article は，大きく分けて環境部と本体部からなり，本体部は通常いくつかの定義と定理から成り立っています.

また，Mizar article では :: 以降はコメントになります.

Chapter 4
証明の方法

4.1 基本的な証明方法

証明の方法について説明します．証明の方法としては，さきほど説明したように，

```
証明したい式
 proof
  . . .
  . . .
  thus (hence) ...;
 end;
```

証明したい式をまず書き，proof（証明），end;（証明終り）の間に証明を書きます．証明の最後の行は必ず thus ... または hence ... で終わります．

p implies q という命題を証明したいとします．

```
p implies q
 proof
 assume p;        p を仮定する
  . . .
  . . .
 thus q;          q が示せた
end;
```

まず証明したい式 p implies q を書き，proof ,end;と書きます．proof にはセミコロンをつけなくて，end にはセミコロンをつけます．p であるならば q ということを証明すれば良いのですが，p というのはこれも 1 つの論理式になっているのです．まず，assume p と書き p を仮定し，それから q が示せたら thus q で証明終りです．

次に，a=c が仮定されいて，b=c implies a=b を証明したいしますと，次のようになります．仮定の a=c には A:というラベルをつけておきます．

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

assume ... で ... を仮定するということになります．仮定部であることを示すわけです．また，hence ... は hence の直前の式と hence の式の理由部からこの証明の最後の式がいえたということを示します．thesis とは,示すべき式ということです．

thus と hence をもう少し詳しく説明しましょう．

thus と hence は証明中で使いましたが，証明すべき式が thus と hence があらわるたびに証明済の部分が & でつながった式よりそぎ落とされて簡単になっていきます．例えば,

```
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;
```

は and でつながった 3 つの式の証明です．最初の hence で 3-1=2 が証明されましたので，5-2=3 & 6-3=3 が証明すべき式となります．次の hence で 5-2=3 が証明されましたので，6-3=3 が最終的に証明すべき式となります．最初に hence thesis で，残りの部分(6-3=3)が全て証明されたことになります．

なお，hence は前述したように then と thus を組み合わせたものです．例えば，

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

というのを，

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

と書けます．

4.2 背理法

また同じことを証明するに，こういうやり方もあります．p implies q を証明したいときに，

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

proof 中で，条件を仮定して assume p，そして結論を否定します．assume not q．そうして，これから導いて thus contradiction(矛盾)．これは背理法を使ったことになります．p でかつ q でないと仮定したら矛盾が生じたということになります．

4.3 証明の分割

証明の構成もいろいろあります．これらを証明のスケルトンといいます．

たとえば，p & q & r を証明したいとします．このとき，次のように証明することも出来ます(この節は 4.1 の繰り返しになります)．

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

まず p が証明できたとします．そうすると thus p となります．また証明を続けて thus q，さらに証明していって thus r．この場合 Mizar のシステムは，thus ... を捜します．これには thus が 3 つもでてきますので，最初の thus で証明すべき命題 p & q & r のうちの p が証明できたということを認識し，次に thus q で 2 番目も証明できたことを認識します．最後に thus r ですべて証明されたことを認識します．最後は thus thesis とすることもできます．この場合，システムは既に p, q は証明されていて，残っているのは r だけで，r が thesis だと認識します．こうして，thus thesis は thus r であるということを認識します．どちらを書いてもかまいません．

それから p & q & r を証明するときに，p を先に証明し，q & r を後に証明したり，あるいは p & q を先に証明し，r を後に証明してもかまいません．自由に分割して証明することができます．

4.4 同値の証明

次に p iff q ($p \Leftrightarrow q$)の証明の仕方ですが，これは p implies q を証明し，それから q implies p を証明します．p iff q は，p implies q & q implies p とシステムは自動的に解釈します．ですから，p implies q をいいますと，前半は証明したことになります．あとは後半の証明だけです．そうして assume q で，thus p となったので，これで p iff q がいえたと判断するわけです．

```
p iff q
 proof
  assume p;
   ...
  thus q;        p ⇒ q がいえた
  assume q;
   ...
  thus p;        q ⇒ p がいえて p ⇔ q
 end:
```

4.5 now の使い方

さらに証明のスケルトンですが，not p を証明するのに，now を使うことがあります．now は次のように使います．

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

now は，now ～ end で 1 つの文(論理式)とみなします．この now ～ end は，not p と同じものです．あるいは，assume p で，p implies contradiction という論理式です．

4.5.1 end についての注意

end についての注意ですが，Mizar では，proof ～ end，now ～ end などで end が出てきて，またこれらがネスティングをなしてきます．end がたくさんあると，どれがどの end かわからなくなってしまいます．ですから now を書いたら end は必要なので，now を書いたら end をすぐに書いてしまいます．あとは，この中にエディターでどんどん挿入していけばいいわけです．同様に，proof と書いたら，必ず end と書いてしまいます．end だけを書くということはないと決めておくと間違いはありません．

4.6 全称記号

次に述語命題にはいっていきます．for x holds p[x] & q[x]の証明をします．なお p[x]とは x を含んだ述語という意味で，かぎかっこが許されているわけではありません．

```
for x holds P[x] & Q[x]
 proof
  let x;
   .....
```

```
      .....
     thus P[x]:
      .....
      .....
     thus Q[x]; (thesis)
    end:
```

全称記号 for に対する言葉として let があります．let x としておいて，thus P[x]で，P を証明します．さらに thus Q[x]で Q を証明します．

それから混乱しなければ，これは文字が変わってもかまいません．例えば y になってもかまいません．

4.6.1 変数が 2 つある場合

それから次の様に変数が 2 つある場合ですが，

```
  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;
```

このように変数を 2 つ書きます．直接的には，$(\forall x)(\forall y)\big(P(x,y)\Rightarrow Q(x,y)\big)$ を証明したことになります．assume .....は 1 つの論理式になっています．

4.6.2 such that を使った書き方

それから，もう 1 つの書き方として次のようなものがあります.

```
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;
```

なお，such that のあとに，ふつうはラベルを入れます(この例の場合 L). そして，最初の行のは st で，3 行目は such that です.

4.6.3 文の分割

こういうスケルトンがあります.

```
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:
```

これを次のように書き換えることもできます. 注意しなければならないのは & と and の違いです.

3 行目の,

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

は,

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

というふうに，書くこともできます．前の書き方と後の書き方は原理的には同じです．前の書き方の and は，別のラベルをたてるために 2 つの文に分けるためのものです．これは，A & B という 1 つの文を，A という文と，B という文とに分けたということです．それに対して，後の書き方はあくまで 1 つの文で，論理演算子でつながったものです．このように，and を使って 1 つの文を 2 つの文に分けたり，逆に&を使って 2 つの文を 1 つの文にすることもできます．

4.6.4 文の分割 assume での場合

such that 後の書き方の違いですが，これと似たことは assume の場合にも起こります．例えば assume の例に，& S[x, y]を加えて，

```
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;
```

と，なったとします．

このとき，

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

は,

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

と書くこともできます．後の書き方に限って，P[x, y]と S[x, y]の両方にそれぞれにラベルを書くことができます．この例では L1，L2 というラベルです．

4.6.5 that の後の注意

that の使い方についていいますと，that の後には then を使うことができません．どうしてかというと，such that の後に 2 つ以上の文があるかもしれないと考えるからです．2 つ以上の文がある場合は，直前の文というのがはっきりしません．ですから，

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

こういう書き方はできません．必ずラベルを使って，

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

こういうふうに書かないといけません．あるいは，B だけ使うときは，

```
C by L2;
```

これだけ書きます．引用が 1 つの場合にも then は使えません．then でなく by L2 と書かなければいけません．ですから that が来たらラベルが来なければこの文は使えません．that の後には必ずラベルが来ると考えられます．

that をつけなければ then を使えます．ですから，最初の書き方，
assume P[x, y] & Q[x, y]の場合は，

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

と使うことができます．that がなくて，1 つの文しかないわけです．assume のときは then を使います．

4.7 存在記号

さらにいろいろなスケルトンがあります．存在記号を含んだ述語論理式 ex x st P[x]の証明のしかたですが，この場合 take というものを使います．

take を使って，

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

と書きます．P[x]を満たす x が存在することを証明するときに，P[a]が証明されたとします．すると take a として，存在する x として，a を取ってくれば，L による証明をされたことになります．

4.7.1 consider の使い方

take と逆の言葉として consider があります．

```
(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;
```

となります．まず，assume します．assume ex x st P[x];として，ここで consider を使って then consider a such that L:P[a]とします．この場合も such that がきますから，ラベルがないといけません．そして，こうしておいて，次に fory st ..... を言うために let y such that Q[y]として thus Q[y]が言えればいいわけです．

consider とは，P[x]を満たすような x が存在するとき，その存在する x に，なにか固有名詞を与えるという意味です．だから存在する x をいま 1 つ話題にしましょうということです．ですから，x でなくてもかまいません．a でもいいです．

存在する a として，ここでは a が固有名詞のように使えるわけです．どこかに存在するとして使われているわけです．ですから consider は take の逆になります．take は固有名詞で何かにおけることに，存在記号をもって来ることに使います．

4.7.2 given の使い方

それから,ちょっと変わったスケルトンがあります．

```
(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;
```

この中で2行目の given x such that P[x]; の given は assume と consider を合わせた働きがあるわけです.

```
given = assume + consider
```

と覚えてください．こちらの方が簡単です.

4.8 場合分け

次に場合に分けて証明するときの話をします．証明したいのは A and B という命題で，この命題は場合分けすれば証明できるとします．例えば x=1 の場合と x=2 の場合と，x がその他の値を取る場合とに分けて証明したいとします．x=1,2 で特異点になっているということです．このときには次のような証明の構造をとります.

```
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;
```

now per cases ～ end の中で，それぞれの場合について case を分けて証明します．case x=1 から最初の thus A and B までで x=1 のときの証明をします．同様に，x=2 の時，その他の場合についても証明します．このとき，now から end までの文は全体で 1 つの A and B とみなします．例えば now ～ end の後に then とすれば，then は now ～ end 全体を受けるわけです．case でそれぞれの場合に分けられていますが，それぞれの場合の帰結の部分を最後に渡すわけです．ですから，これ全体を引用したときには case に分かれたということは忘れてしまって，結論だけが全部同じということがチェックされるわけです．だからこれは 1 つの式だということです．なお，L のような式は論理的に明らかなので，now per cases by L の by L はなくてもいいのです．

4.8.1 場合分けの前提

ここで注意があります．この場合は，x=1 or x=2 or not (x=1 & x=2) ---(1) という式を場合分けするときに前提として使っています．この場合分けがすべての場合を覆っているということをいっているわけです．このように場合分けするときは，すべての場合についていっていることを言わなければなりません．

なお，上の例のように(1)式の成立が明らかなときには，now per cases の後の by は不要です．

4.8.2 場合分けでのラベル

当然ながら場合分けをして証明するときには，それぞれの場合の証明でのラベルは，その場合の証明の中でのみ有効です．もし，その場合の証明以外でその証明のラベルを使おうとすると，ラベルがないというエラーとなります．

4.8.3 ラベルについての注意

それから，ここでラベルの注意をしておきます．Mizar では，同じラベルを使ってもかまいませんが，直前のものが優先されます．

例えば，

```
L1:......;
    ......
L1:......;
    ...... by L1; ここでは直前の L1 が引用される
```

この例で何か証明したときに by L1 とすると，下の L1 の文が引用されます．

このことは間違いやすく L1，L2，L3，...... とラベルをつけていって，どこまでラベルを使ったのかを忘れてしまい，また L3 とつけたとします．そして，by L3 とすると，自分では前の L3 のつもりで必ず L3 から推論できると思っても，いくらやってもエラーがでてきてしまうわけです．このようなことを最初のうちによくやります．このときはよく捜すと，同じラベルが 1 つ隠れているのです．もうひとつ別のところにあることがあります．

4.9 @proof の使い方

Mizar で長い article を書いていると，くりかえし Mizar チェッカーでチェックすることになります．長い article をチェックするには時間がかかります．このようなとき，既に証明が終わった定理などはチェックすることを省略することによって，Mizar チェックにかかる時間を節約することができます．

定理などのチェックするのを省略するには，proof を@proof に変えて置きます．こうすることにより，Mizar チェッカーはその定理をチェックしません．

もちろん article が完成したら，すべての@proof は取り除かなければなりません．

4.10 新しい変数の導入(Set,reconsider)

新しい変数を導入する場合,

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

のように書きます．これによって新しい変数 r1 を(r + 2)として導入できます．その型(type)は r が実数型であれば，実数型になります.以後の文のプロックにおいて，変数 r1 を自由に使うことができます (end 文によってそのブロックを抜けると r1 は無効になります)．型の異なる新しい変数を導入したい時は，reconsider 文を使います.

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

のように書きます. この場合 r + 2 が自然数 (Natural Number) 型となる理由付けが必要になります. そのブロックの中でだけ有効であることは Set 文と同様です.

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

のように 2 つ以上の変数を同時に導入することもできます.

4.11 take について

4.7 節の繰り返しになりますが，take についてより詳しく説明します.

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

を証明するとき，次のようにします.

```
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:
```

最終の式が(1)の形でなく

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

となっていることに注意して下さい．これは途中に take 3 があるためで，これによって最終の式から存在記号の部分が取り除かれ，裸の部分だけを証明すればよいことになります.

これをもう少し詳しく説明すると，

```
(∃x)(∃y)(f(x,y))を証明するのに,
proof  ------ ------ この段階での証明すべき式は(∃x)(∃y)(f(x,y))
        ------ ------
       take a  ------ この段階での証明すべき式は(∃y)(f(a,y))
        ------ ------
       take b  ------ この段階での証明すべき式は(f(a,b))
        ------ ------
       hence f(a.b);
end;
```

だから次のような証明ができます.

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

なお，take を使わなくても多くの場合，上の f(a,b)にあたる所を証明すれば，最終的にチェッカーが a と b を探してくれます．例えば,

```
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;
```

のようにしてもよいのです.

4.12 hereby

証明中で使われる hereby は thus + now と同じです.
例えば,

```
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;
```

では, (for x st x in A holds x in B)の命題がまず証明されて, 証明すべき命題群から除かれます. 即ち,

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

の部分が，上の(for----)と同じです．これがまず証明される．ということは，

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

というようにthusがnowの前につけられます．これは，

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

となるのです．

この応用編として，次のようなherebyの使い方があります(これは証明をつけるまでもなく自明な定理ですが)．

```
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:
```

(x=y iff not y<>x)という命題は，(x=y implies not y<>x) & (not y<>x implies x=y) という命題と同じです．hereby で前の(-----)の部分が証明されて取り除かれ，後半では，後の(-----)の部分を証明するのみ，となります．

更なる応用編として，

```
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;
```

があります．これは，環境部の definition 部に BOOLE があることによって，(A=B) は，(A c=B)&(B c=A) と同じとみなされます．更に definintion 部に TARSKI があることによって，これは，

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

と同じとみなされます．この形になれば，上の例が hereby で記述できることが分かるでしょう．実際には，この最後の例のように，2 つの集合が等しいことを言うときに，よく hereby が使われます．

Chapter 5

mode について

5.1 mode

次に mode (モード) についての話をします．Mizar には mode という概念があります．mode とは，Pascal (コンピュータ言語の一種) などでの型 (タイプ) に相当するもので，一般的な mode としては set があります．その他に代表的な mode として Nat, Real などがあり，またユーザが mode を定義することができます．また mode は図の様に set (すべて)の中に Real (実数)があって，さらにその中に Nat (自然数)があるというふうに階層構造をなしています．そして，functor などは mode に従属しています．このことはプログラミング言語と似ています．なお set と Any は同じとみなされます．

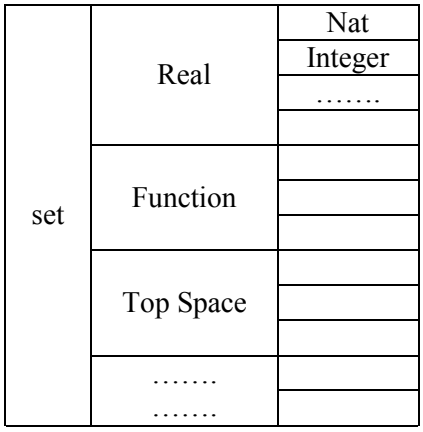


図 5.1 mode の階層

例えば，次のような例を考えてみます．ここでは + は Real についてのみ定義されているとします．

```
reserve A,B for Real;
        C,D for Function;
```

```
L:C=A+B:
 ......
 D=A by...;
 then C=D+B by L;
```

A,B が Real，C が Function のときに，C=A+B と D=A が証明されたとします．ここで D=A という式を C=A+B に代入して，C=D+B by L とするとるとエラーになります．どうしてかというと，これは代入できるはずですが，D と B は mode が違うわけです．+という演算子は Real に対してのみ定義されているのであって，Function については定義されていないわけです．ですから，unknown functor というエラーになります．

このエラーは変数 D の mode を拡張することにより解決することができます．Function である D を Real でもあることがいえればいいわけです．

5.2 階層構造

さて，最初に言ったように，mode は次のような階層構造になっています．set は一番広い mode で，例えばその下に Real，さらにその下に Nat があります．Nat なら set, Real なら set, Nat なら Real となるわけです．ここの考え方は，オブジェクト指向言語 (Object Oriented Language) の考え方と似ています．同じ記号を，上の方に使っていくわけです．

5.3 mode の変更

このような場合，reconsider という言葉を使って，mode の変更を行います(4.10 参照)．

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

このように reconsider によって，Real の mode である D を Nat の mode である U に変更することが出来ます.

5.4 証明中の mode の宣言

証明中に変数の mode を宣言することがよくあります．また証明のスケルトンに入りますけれども，例えば P⊆Q の証明で,

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

のとき, 3 行目で let x be set; としています.

be mode 名

で証明中に mode を宣言することもできます．もちろん，前もって reserve x for set とすることも出来ます.

5.4.1 c= についての注意

また，注意ですが，Mizar では⊆の代わりに c= を使いますが，c= を使う場合は c= の前後に 1 つづつスペースをあける必要があります.

5.4.2 実際に definition を使っている例

それから，ここでは，$(\forall x)(x \in P \Rightarrow x \in Q)$ を証明していて，$P \subseteq Q$ を証明したわけではありません．両者のあいだには数学的距離があります.
しかしこれは 3 章で説明したように環境部の definitions の中に TARSKI を書いておくことにより，システムが自動的に両者が同値であることを理解します．これは，TARSKI の abstract ファイル中に predicate(述語命題)としてこのことが定義されていることによります.

TARSKI には，Mizar ライブラリの基本となる定理から構成されているため，アーティクルを書くにあたっては環境部の definitions のところに TARSKI を書いておきます.

Chapter 6

定義

6.1 定義

ここでは定義について説明します．predicates，functor，mode の定義について説明します.

6.2 vocabulary file の書き方

Mizar では，新たに functor を定義するときには，その functor を vocabulary file に登録する必要があります．predicates, mode を定義するときも同様に登録する必要があります．vocabulary file (.voc ファイル)に，必ず行の先頭から書き始め，最初の 1 文字に何の名前なのかを書きます．例えば，HIDDEN.VOC ですと，

HIDDEN.VOC
MAny
MReal
MNat
K[:
L:]
O+ 32
O.
R<>
R<
R>

（一部抜粋）

となっていて，行の先頭の 1 文字で何なのかを示しています.

記号	定義するもの	意味
M	Mode	モード
O	Functor	関数記号
R	Predicate	述語命題
K	Left functor bracket	左括弧
L	Right functor bracket	右括弧
G	Structure	構造
U	Selector	識別子
V	Attribute	属性

行の先頭の 1 文字に続けて，スペースを空けずに名前を書きます．さらに functor の場合は，そのあとにスペースを空けて，優先順位を書きます．優先順位は 1 から 255 までの整数で．数字が大きいほど優先順位が高くなります．省略すると 64 となります．

こうして vocabulary file に登録してから，article で定義します．なお vocabulary を新たに登録する場合，既に登録されていないかどうかを確認するために，

```
CHECKVOC [vocabulary]
```

として確認してください．

6.3 predicate の定義

まず，predicate の定義について説明します．例えば，k, l を自然数として，いま，k は l の因子である，という定義をしたいとします．これは predicate です．`k is factor of l`，このような定義をしたいとします．これは二項演算子の様な形をしてますけれども，k, l と 2 つ変数を持っているわけです．この場合どうやって定義を書くかというと，

```
definition
  let k,l be Nat;
  pred k is_factor_of l
  means
  :FACTOR:
```

```
ex t being Nat st l=k*t
```

となります. let k,l be Nat の Nat は, k, l の Mode が自然数であることをあらわしています. :FACTOR:はラベルで, 後で引用するときにこのラベルを使います. 定義のなかのラベルは両側をコロンではさみます. そして, これは abstract file をここから抜きだして作るときには, def に数字がついたものに変えられます. ふつうの定理のラベルは単なる数字に変えられます. また k*t の * はかけ算です.

これで, k is_factor_of l という predicate とは ex t being Nat st l=k*t ということを意味しています.

6.4 functor の定義

次に, functor の定義ですが, 例えば 2 変数の functor を考えましょう. 集合 (set) X,Y から, Z という新しい集合を作りだしたいとします. このときどうするかという話をします.

$$X /\backslash\ Y \to Z$$

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;
```

この場合も，$f(X,Y)=Z$ という関数定義でなく，演算子の形で書けます．
$Z=X \cap Y$，こういう記号を定義したいとします．集合 X と集合 Y からあたらしい集合 Z を作り出す，それがまさにこの functor です．`let X, Y be set;` で，集合 X, Y を用意します．`func X /\ Y → set` で，`X /\ Y` が `set` であるということを表しています．`means` の次の行が，その functor の意味です．先程と同じ様にラベルをつけて，functor の意味を書きます．この意味中で，`X /\ Y` を `it` と置きます．

そして，証明部(意味の内容)では，`existence` (存在)と `uniqueness` (ユニーク)の証明をしなければいけません．存在して，かつユニークであることをいわなくてはいけないわけです．

`uniqueness` の証明のしかたですが，以下のことを言えばいいわけです．

`it` として `A1,A2` を取ってくると，

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

これを書くと，

```
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;
```

というふうになります．つまり，この条件をみたす集合 2 つ A1, A2 をとってくると，実はそれらは等しいということを証明することで uniqueness を証明したことになります．

この uniqueness の証明の中で，まず，3 行目に such that がありますから，そのあとに 2 つの文，ラベル A1 文とラベル A2 文が and で結ばれていることがわかります．そして now let y となっています．この y は set であることが reserve されているとします．そして，ラベル A1 より，y in A1 iff y in X & y in Y であることが言えて，さらに，このこととラベル A2 より，y in A1 iff y in A2 であることが言えるわけです．そして，A1=A2 であるための必要十分条件は任意の y に対して，y in A1 と y in A2 が同値であるという，TARSKI:2 より，A1=A2 が証明されたわけです．

existence も同様ですが，こういう it が存在するということを証明します．すなわち，

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

を証明します．

6.5 equals の使い方

equals を使うと，$Q(x)=x^2$ などのように，置き換えによる関数の定義が簡潔に行えます．例えば，

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

のようにすれば，関数 Quard(x)=x*x が定義できます．

6.6 mode の定義

最後に mode の定義について説明をします.

いま，あらたに nlt2_elements という mode を定義するとします.

```
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;
```

mode 定義の証明部では，existence についてのみ証明が必要となります．また mode が定義されると，その性質を(attribute)や(cluster)で定義する事ができます．例えば,

```
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;
```

一般には，attribute は次のような述語の形か,

```
A is non-empty
```

形容詞の形で定義します.

```
non-empty set
```

この場合，non-empty は，non empty と同様です.

また，attribute の definition ブロックの中で，それと同じ意味を持つ シンボル (synonym) と反対の意味を持つシンボル (antonym) を定義する事ができます．例 (これは実際のものではない) :

```
definition
 attr empty -> set means :A1:
  not ex x st x in it;
  ...
 synonym (void);
 antonym (non-empty);
end;
```

一つの mode に複数の attribute をつなげて使いたい時，その cluster を定義する必要があります．その場合，存在の証明が必要です．そうしなければ，次のような矛盾するコンビネーションが現れます．

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

このように，Mizar では新しい functor，mode，attribute，cluster (§6.7 参照) などの定義が出来ますが，それぞれの証明は以下の要素を含む必要があります．

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

6.7 Cluster の定義

ある mode に属性 (attribute) をいくつか附与して，新しい mode のようなもの (タイプ) を作り出すことができます．例えば，Real という mode があるとします．この mode に対して positive という属性が定義されたとしましょう．このとき，positive Real という mode (これを cluster という) も作ることができます．

それなら positive_Real という新しい mode を作ればよさそうですが，cluster が便利なのは，

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

はいえませんが，

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

は理由付けなしにいえることです．即ち，"positive Real"という cluster は positive な Real であることを，Mizar チェッカーがしっかりと覚えていてくれるのです．一度新しい cluster が定義されれば，他の articles で，environ 部の clusters のというように最初のファイル名を記入しておけば，そこで，その cluster を自由に使うことができます．

clusters を定義する例を下に掲げます．その定義には existence だけを証明する必要があることに注意しましょう．

なお，VOC ファイルには，あらかじめ

```
Vpositive
```

と書かれているものとします．

```
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 0<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 structure の導入

(X,a) を二項代数とする，というような場合(X は集合,a は関数),

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

とだけ書きます(definition　など，つけないで). 新しい用語 Binalg, base, op2 のためには VOC ファイルの中に,

```
GBinalg
Ubase
Uop2
```

と書いておきます(G, U の種別コードをつけることに注意).

すると次のような定理は理由付けなしに成立します.

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

これは，「二項代数 (X,a) のベースは X である」という命題に対応します. base に the をつけることに注意して下さい.

Chapter 7
実行環境

ここではMizarのシステムを自分のパソコンにインストールして利用するときの実行環境について説明します．実行環境としては，Windows9x/2000/NTのMS-DOSプロンプト（コマンドプロンプト）で実行するMS-DOS版のものと，intel版のLinux（kernel 2.0.x, 2.2.x, 2.4.x）で実行するLinux版のものがありますが，いずれの場合もなんらかのEditorが必要です．Mizarでは拡張文字を使いますので，拡張文字の表示できるEditorを用意してください．MS-DOS版では標準で付属しているEditorがお勧めです．そして，複数のライブラリファイルを参照しますので，複数のエディタウインドウが開けるものが便利です．

7.1 インストール

Mizarの実行環境インストール用のファイルはMS-DOS版，Linux版とも約14Mバイトの大きさのアーカイブファイルです．したがって， Mizarの Homepageや ftpサーバからブラウザや ftpコマンドを使って入手するのが簡単です．たとえば Homepegeであれば次の URLなどから入手してください．

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

anonymous ftpサーバからの入手については Appendix Cを参照してください．

入手したMizarのアーカイブファイルはハードディスクにインストールして使用します．

7.1.1 MS-DOS 版

MS-DOS 版の Mizar は Windows9x/2000/NT の動作する AT 互換機のハードディスクにインストールして使用できます．なお約 75M バイトのハードディスクの空き容量が必要です．ここでは，ハードディスクが C ドライブの Windows98 の PC に Mizar のアーカイブファイル（例えば `mizar-6.1.11_3.33.722-win32.exe`） から実行環境の構築する流れについて説明します.

Mizar のアーカイブファイルは，ハードディスク内の適当な場所（たとえば C:\work など）にディレクトリを作成し，そこに保存します.

Windows のスタートメニューから MD-DOS プロンプトを立ち上げて，us コマンドで英語モードにします．そしてカレントディレクトリを先ほどの Mizar のアーカイブファイルを保存したディレクトリに変更します.

Mizar のアーカイブファイルは自己解凍ファイルになっていますので次のように入手したファイルのファイル名を入力し解凍します.

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

解凍したファイル郡の中に INSTALL.BAT というインストール用のバッチファイルがありますので，これを次のように使い Mizar のインストールを行います.

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

`C:\MIZAR` は Mizar のシステムがインストールされるディレクトリですが，他のドライブや他のディレクトリ名を指定することもできます．このバッチファイルにより Mizar のコマンドがインストールされ MIZ ファイルや今まで作られてきた abstract file などもディレクトリが自動的に作成されてインストールされます.

インストールが終わったら，コマンドサーチパスに `C:\MIZAR` を追加します．また，Mizar のインストール先のディレクトリに `C:\MIZAR` 以外を指定した場合は， システムの環境変数 MIZFILES を Mizar のインストール先のディレクトリにします.

具体的には，Mizar のインストール先を `C:\MIZAR` にしたとき autoexec.bat には次の 1 行を加えたのち PC を再起動します.

```
PATH C:\MIZAR
```

なお，autoexec.bat ファイルに，たとえば

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

など既に PATH の指定がある場合，セミコロンで区切って次のように Mizar のディレクトリの指定を加えます.

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

また，例えば Mizar のインストール先を D ドライブのハードディスクの `D:\MIZAR` にしたとき autoexec.bat には次の 2 行を加えたのち PC を再起動します.

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

なお，Mizar のアーカイブファイルを解凍したときにできた readme.txt というファイルにこれらの詳しい説明がかかれていますので必要に応じて参照してください.

7.1.2 Linux 版

Linux 版の Mizar は Linux(kernel の 2.0.x, 2.2.x, 2.4.x で動作確認済み)がインストールされた Intel の Pentium プロセッサを持つ AT 互換機のハードディスクにインストールして使用できます．約 80M バイトのハードディスクの空き容量が必要です．ここでは，PC に Mizar のアーカイブファイル 例えば (`mizar-6.1.12_3.35.723-linux.tar`) から実行環境の構築する流れについて説明します.

Mizar のアーカイブファイルは，ハードディスク内の適当な場所（たとえば自分の home ディレクトリ上） にディレクトリを作成し（たとえば `work`），そこに保存します.

そしてシェルのコマンドプロンプトでカレントディレクトリを先ほどの Mizar のアーカイブファイルを保存したディレクトリに変更します.

そこで Mizar のアーカイブファイルを tar コマンドで展開します．#がプロンプトとすれば

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

とします.

展開したファイル郡の中に install というインストール用のシェルスクリプトがありますの
で，これを次のように使い Mizar のインストールを行います.

```
# ./install
```

デフォルトでは Mizar の実行ファイルは /usr/local/bin に，Mizar の shared ファイルは /usr/local/share/mizar に，Mizar のドキュメントファイルは /usr/local/doc/mizar にインストールされます．したがって，インストールのときはこれらのディレクトリに対して書込み権のあるユーザーで作業を行う必要があります.

デフォルト以外のディレクトリにインストールすることもできます．この場合はインストール用のシェルスクリプトをまず起動してインストール中に画面上で対話的にディレクトリを指定していきます.

インストールが終わったら，システムの PATH にインストールした Mizar の実行ファイルのディレクトリが含まれていることを確認してください．デフォルトの場合であれば,

```
# echo $PATH
```

と入力すれば /usr/local/bin が PATH に含まれていることを確認できます.

もし含まれていない場合には，たとえば .bashrc (bash の場合)に

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

をくわえるなどしたのち

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

を実行してコマンドサーチパスに /usr/local/bin を追加します．なお，デフォルト以外のディレクトリを Mizar の実行ファイルのディレクトリに指定したときは当然そのディレクトリを指定します.

次に環境変数 MIZFILES に Mizar の shared file のインストール先のディレクトリを指定します.

具体的には，Mizar の shared file のインストール先が /usr/local/share/mizar であれば（デフォルト），たとえば， .bashrc (bash の場合)に

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

をくわえるなどしたのち

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

を実行します.

なお，Mizar のアーカイブファイルを解凍したときにできた README というファイルにこれらの詳しい説明がかかれていますので必要に応じて参照してください.

7.2 Mizar を使うための準備

Mizar を使うためには，まず各自のディレクトリを作る必要があります.

ここでは MS-DOS 版に準じた一例を説明しますが，C:\ に USER というディレクトリを作り，その下に各自のディレクトリを作ります.　例えば，著者(中村)でしたら，nakamura というディレクトリを作ります．そしてその下に，TEXT，DICT，PREL の 3 つのディレクトリを作ります.

それぞれのディレクトリについて説明しますと，TEXT はユーザが書いた Mizar article(.MIZ ファイル) を置くディレクトリです．Mizar のシステムが出力する中間ファイルもここに置かれます．DICT は，DICTionary の略で，ここにはユーザが書いた vocabulary file (.VOC ファイル) を置きます．新しい言葉はこの vocabulary file の中に書いておきます．PREL は，PRELiminaries の略で，Mizar ライブラリに登録されてない article を参照するときに，ここに必要なファイルを置きます.

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

Mizar を使うときは通常，C:\USER の下の自分のディレクトリをカレントディレクトリとして使います．たとえば 7.4.1 で説明する accom という Mizar のコマンドを使うときは

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

のようにします．

7.3 Mizar のシステム

Mizar のシステムは，MS-DOS 版では， C:\MIZAR に作られますが，その中で大事なディレクトリとして，ABSTR があります．これは，ABSTRact の略で，この中に今まで作られてきた abstract file があります．例えば，BOOLE.ABS や， TARSKI.ABS です．article を書くときに，この中のファイルを頻繁に参照します．

その他のディレクトリとしては以下のものがあります．

PREL には，Mizar ライブラリに登録されている article を参照するときに使うファイルが置かれています．

MML にはライブラリの MIZ ファイル（詳細な証明が付いているもの）がインストールされます．

DOC には インストールした Mizar の README ファイル，ライブラリに登録する時に使うドキュメントやその他 Mizar に関連するドキュメントがインストールされています．

Mizar の様々なコマンドは C:\MIZAR 直下に置かれます．

同様に Linux 版では，7.2.1 のインストール手順の説明でも書きましたように，Mizar のさまざまなコマンドは /usr/local/bin にインストールされ，ABSTR, PREL や MML のディレクトリは /usr/local/share/mizar に，DOC は /usr/local/doc/mizar に作られ，ファイルがインストールされます．

7.4 Mizar の使い方

7.4.1 accommodate

例えば EXAMPLE1.MIZ というファイル名の article を作ったとします．このファイルは 7.2 で作成した各自のディレクトリの TEXT というディレクトリに置きます．まず，accommodate(アコモデート，ファイルを集めて調整するの意)します．accommodate するには，コマンドプロンプトで ACCOM いうコマンドを引数として EXAMPLE1.MIZ を指定して，

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

とします．引数として指定するファイル（ここでは EXAMPLE1.MIZ）はパスを含めて指定しなければなりません．accommodate では，環境部を読みこんで，その article で必要な，参照している article の定理，定義などを記号化したいくつかのファイルを作り，article の証明のチェックをする準備をします．ここでエラーが出ることもあります．例えば，環境部に abstract file のファイル名を誤って書いてしまった場合などです．

accommodate は，毎回実行する必要はありませんが，環境部を変更したときは accommodate しなおさなければなりません．

7.4.2 Mizar チェッカー

これが通ったら，次に Mizar チェッカーを実行します．Mizar チェッカーを実行するには，

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

とします．すると，画面に表がでてきて，Parser, Analyzer, Checker の順で，article をチェックし，チェックしている行数と，エラーの個数が表示されます．チェック後にチェックした行数だけが表示されていれば，エラーが一つもなく，Mizar のチェッカーに通ったことになります．

7.4.3 エラーがある場合

エラーがある場合，ERRFLAG というコマンドを使っても，エラー番号を article にコメントの形で記入できます.

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

とします．いずれのコマンドも拡張子 .MIZ は省略することができます.

エラーメッセージを見るには，別のファイル (MIZAR.MSG) をエディターで見る必要があります.

7.4.4 便利なコマンド

便利なバッチコマンドとして，MIZF.BAT があります．コマンドラインで

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

とすると，accommodate する必要があるかどうかを調べて，必要なときには accommodate を実行し，そのあと Mizar チェッカーを実行します．さらにエラーがある場合，エラー番号をコメントの形で article に書きこみ，article の最後には発生したエラー番号とエラーメッセージの対照表が書きこまれます[1].

---

[1] 書き込まれたコメントは，次に MIZF コマンドを実行したときに取り除かれます.

Chapter 8
ライブラリへの登録

Mizar ライブラリへの登録について説明します．Mizar の article を書いて，チェッカーを通ったとします．しかし，チェッカーを通っただけでは，Mizar ライブラリに登録することはできません.登録する前にいくつかやることがあります.

8.1 article の改良

8.1.1 本体部の改良

まず，article の改良をしなければなりません．このためのコマンドとしては,

```
RELPREM     余計な引用をチェックする．
CHKLAB      余計なラベルをチェックする．
INACC       余計な行をチェックする．
TRIVDEMO    トリビアルな証明をチェックする．
RELINFER    余分な行をチェックする．
```

というコマンドが用意されています．いずれのコマンドも,

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

のように使い，それぞれのコマンドを実行した後に,

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

を実行すると，先ほどと同様に，article にエラー番号が書きこまれます.

RELPREM は，by の後ろに推論に関係ないものが引用されているかをチェックします. また，then についても，それが必要であるかどうかチェックします．もし，それがなくても推論できるようなものがあると，エラーになります.

CHKLAB は，ラベルをつけたが，あとで使ってないラベルをチェックします. INACC は，あとで引用されてない行をチェックします．もし，このチェックでエラーがでて，行を削除した場合，それにより余計な引用やラベルができるかもしれませんので，また最初からチェックしなければいけません.

TRIVDEMO は，トリビアルな証明があるかどうかチェックします．これは，

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

となっているような場合に，

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

と簡単化できることを示します． T1 と T2 は上の proof と end の間で使われているラベルです(proof --- end の外部のラベル).

RELINFER の説明

RELINFER は 2 つの行を 1 つにまとめるかどうかをチェックします．例えば，

```
A=B + C by T1;then
D=E + (B + C) by T2;
        *605
```

などとエラー(*605)が表示されたときは，

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

とすると，一行で済ませられることを示します.

*604 という番号のエラーは,

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

となるとき,

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

と書けるということを示します．もしラベル S1 の行が他に使われていなければ，それを削除することができますが，使われていれば，削除できず，行を減らすことはできません．使われているかどうかは，もう一度 CHKLAB を使えばわかるのですが，使われている時，また元へ戻すのはやっかいなので，*604 は無視してもよいでしょう．いずれにしても，一連のチェックプログラムを何度も用いる必要があります．その結果何割も行数が減ってしまうこともあります．行から行の導出が強力で MIZAR チェッカーにとっては当然でも，人間が読んだときは理解できないような飛躍したものになってしまうこともあります．従って必ずしも行数を最少にまでする必要はなく，ほどほどに留めてもかまいません.

8.1.2 環境部のチェック

そのほか環境部のチェックがあります．このためのコマンドとしては,

IRRVOC	不要な vocabulary をチェックする.
IRRTHS	不要な theorems をチェックする.

があります．使い方は本体部のチェックのコマンドと同様です．NOTATION や CONSTRUSTORS に関しては，良い方法がないので，NOTATION や CONSTRUSTORS は最初からたくさん使わないで，なるべく最低限のもの，最初は 0 から出発して，だんだん自分に必要なものを入れていってください．但し，IRRVOC で不要なファイルを削除しますと，その後 accommodation を行うと NOTATION 部にエラーが出ますので，それを取り除くことはできます.

8.2 登録の準備

article の改良のチェックがすべて終わったら，次に，MIZ2ABS を実行すると，まず accommodate し，そして自分の article (.MIZ ファイル) から，その article が参照されるためのファイルを作ってくれ，さらに Theorem と Definition だけ抜きだして，abstract file (.ABS ファイル) が作成されます．さらに，その article を参照できるように，MIZ2PREL を実行すると，自分のディレクトリの PREL に必要なファイルをコピーしてくれます.

article 改良後に使うコマンド

MIZ2ABS　abstract file をつくる.
MIZ2PREL　article を参照できるようにファイルをコピーする.

そして，その article が引用されるときのために，article を書いた著者名，所属，住所，引用文献などを普通の文章で書き，これを.BIB ファイルとします．DOC ディレクトリの中に example.bib というサンプルがありますので参考にしてください．そうしてできた article を Mizar Society に，たとえば，フロッピーディスクで送ります．投稿された論文はまず機械的にチェックされ，査読委員会で，いい論文(article) と判定されれば，accept され，その通知がきます．accept されますと，Formalized Mathematics という雑誌に Mizar article が自動的に英語の論文に変換されて掲載されます.

またメールの貼付け機能などを使って mml@mizar.uwb.edu.pl へ送ることもできます．メールで投稿するときの方法が Appendix C.3 にありますので参考に

してください.

1)注: SUMMERY の中で，ある論文を引用したいときは，文中で `/cite{A1}` と書きます．別に REFERENCES の記入欄中にその論文名を書き，A1 というコードを与えておきます.

2)注: その中にある PRESENTER というのは，論文のレフリーの主だった人，という意味ですが，空白にしておいてください.

Chapter 9

便利なツール

9.1 用語のある VOC ファイルを探す (findvoc)

新しい用語 (terminology) を導入するには，あたらしい VOC ファイルを作りその中に登録します (6.2 参照) がすでに登録された用語がどの VOC ファイルにあるのかを知りたい時は,

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

とすると，"Real" という文字列を含んだ用語全てについて，それが属する VOC ファイルのリストが表示されます.

```
findvoc
```

とだけ入力すると，findvoc の使い方(ヘルプ)が表示されます．例えば，文字 " | " を探したいときは,

```
findvoc \b
```

そして文字 "く "を探したい時は

```
findvoc \l
```

とすべきことなどが表示されます.

9.2 VOC ファイルの中味を知る

MIZAR の昔のバージョンでは，全ての VOC ファイル が実際に存在したのですが，現在は統合されていて，個々に利用者が見ることはできません．例えば，INDEX1 という VOC ファイルの内容を知りたいときは,

```
LISTVOC INDEX1
```

とすると INDEX1.VOC の内容が表示されます.

9.3 その用語が使用されている ABS ファイルを知る

9.3.1 grep を使う方法

その用語がライブラリのなかのどこで使われているか知りたいことがあります．それは，(どこで定義されているのか)とか，(それに関する定理はどことどこにあるのか)といった疑問のあるときです．そのようなときは，標準の MIZAR システムには入っていないのですが，egrep というプログラムを使うと便利です．例えば Metric という用語があらわれる ABS ファイルをリストアップするには,

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

とすればよいのです．egrep.exe プログラムは，例えば，次の URL(インターネットの)からフリーで入手できます(M.Patnode 氏によるもの).

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

ディレクトリの移動を含めてバッチファイルにしておくと便利です.

また通常の文字列(英数字)を探すときは上記のようで良いのですが，一般にはその文字列を正規表現 (regular expression) で探すことになっています．従って，a+b+c 又は x+y+z のように 3 変数の和があるファイルと行を探すには,

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

のようにしなくてはなりません．ここで

```
[a-z]
```

はアルファベット a から z までの文字のいずれかを表し，\+ は + の文字を表します．

ここで，+ や - のような記号は，バックスラッシュ　\ の後に書かなくてはいけないことに注意しましょう．

正規表現の詳しい説明はインターネット上にたくさん出ています．例えば，

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

を参照してください．

9.3.2 Web から検索する方法

インターネットに接続できる環境にあれば，Web で検索することもできます．

ブラウザで

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

という URL を開いてください．図 9.1 のような検索画面が表示されます．

この画面で調べたい用語をタイプし（図 9.1 では `Metric` ），abstract file から探すのか miz file から探すのか，あるいは両方から探すのかラジオボタンで選択して（図 9.1 では abstract file） search をクリックすることで検索できます．

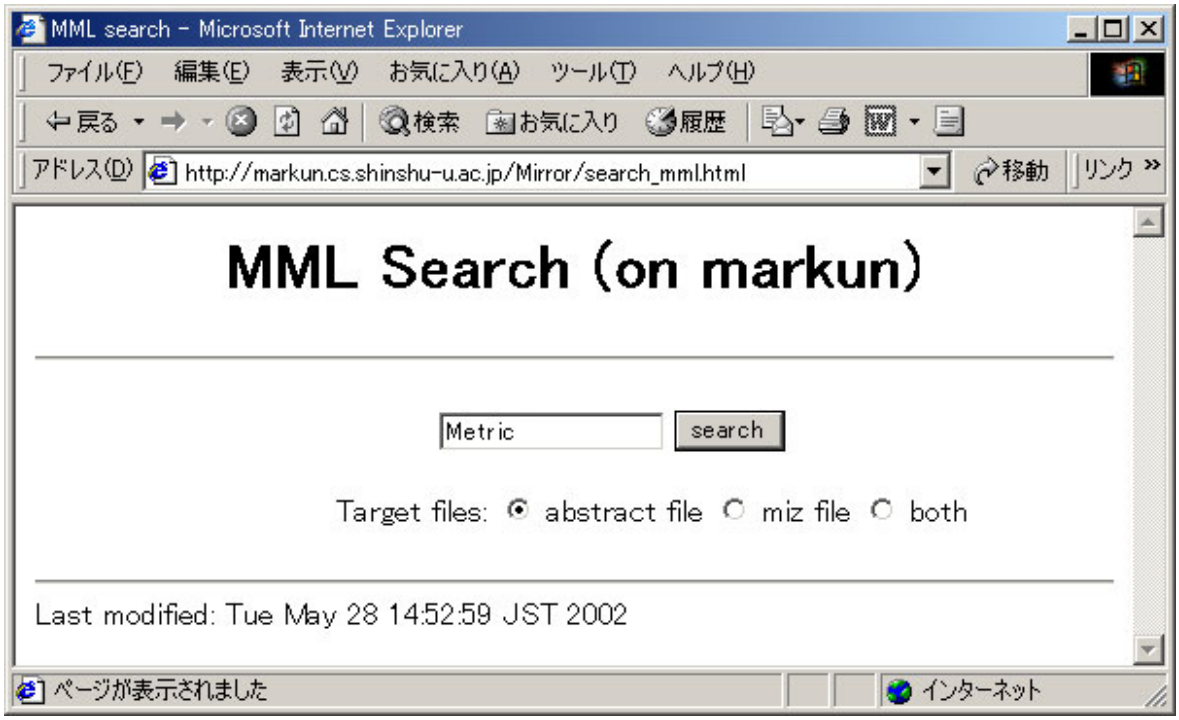


図 9.1　Web での検索画面

Chapter 10
バージョンアップについて

MIZAR のバージョンアップは頻繁に行われます．このとき，ライブラリの増加は当然のことで，自分の作成中の article のチェックに影響を及ぼしませんが，チェッカーのバージョンアップについては注意を要します．インターネットの MIZAR のホームページをよくウオッチングして，最新のものを入手するよう気を付けて下さい.

時々，従来使っていた基本的な定理が cancel されてしまうことがあり，戸惑いますが，新バージョンのディレクトリ中の

```
CANCELED.DOC
```

の中にその情報が書かれています.

Appendix A

予約語一覧表

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
Mizar ライブラリ定理集

ここには，Mizar ライブラリの中で頻繁に引用される article の定理，定義をコンパクトに編集したものを掲載してあります.

適宜，もとの abstract file を参照してください.

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

などから参照できます.

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);
lO8 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;
11 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 Mizar Society 連絡先

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

郵便番号 380-8553
長野県長野市若里 4-17-1
信州大学工学部情報工学科
情報基礎講座中村研究室内
Mizar Society Nagano Circle

Fax: (026)269-5495
E-mail: kiso@cs.shinshu-u.ac.jp

C.2 ftp による Mizar の入手法

Mizar は，以下に示す日本やポーランドの ftp サーバから anonymous ftp により入手することができます

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/
```

Mizar のファイル名は

MS-DOS 版では　`mizar-X_Y-win32.exe`
Linux 版では　`mizar-X_Y-linux.tar`

となっていて，X は Mizar ソフトウエアのバージョンを示し，Y は MML のバージョンを示しています．たとえば X が 6.1.12, Y が 3.33.722 である
`mizar-6.1.12_3.33.722-win32.exe`
は Mizar の Version 6.1.12, MML の Version 3.33.722 である MS-DOS 版のファイルになります．なお，MS-DOS 版のファイルは自己解凍ファイルになっています.

C.3 インターネットの投稿法

出来上った MIZ ファイル，VOC ファイル，BIB ファイルなどを，メールに貼付して送ります．テキストのまま貼付すると，8bit -> 7bit の変換で不具合が生じる可能性があります．そこで，例えば,

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

などの指定をして送る，あるいはそれらを一般的な圧縮形式のファイル(例えば PKZIP, ARJ, RAR, TAR, GZIP など) に圧縮し，メールに添付して送るといいでしょう．送信先は MIZAR ライブラリ委員会

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

です.

C.4 WWW Homepage

Mizar のホームページ URL は，`http://mizar.org/` です.

ここには，abstract ファイルが html 形式で全て掲示されています．またそこで使われている用語が最初にどこで定義されたか，リンクを辿ることもできますので，大変便利です.

またこれらのミラーサーバが，日本の信州大学やカナダのアルバータ大学などにあります.

ミラーサーバ：

```
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

Mizar article が accept されると，英語の論文に自動変換されて，Formalized Mathematics に掲載されます．これは年に数回，ワルシャワ大学ビァウイストーク分校より発行されています．1998 年末現在，全 7 巻発行されており，これらには Mizar ライブラリに登録されている，すべての article が英語の論文として掲載されています.

Formalized Mathematics につきましては下記にお問い合わせください.

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

Formalized Mathematics の正式名称:

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

あとがき

この原本は，中村が 1992 年 9 月 5 日に信州大学で講義した内容をもとに，渡辺，田中，カワモトが加筆，編集したものです.

当時の Mizar はバージョン 3.29 にもとづいた内容で，一部は Mizar バージョン 4.09 をもとに改訂してありました.

本書の内容は， Mizar バージョン 5.2.12 を参考にしている改訂版，バージョン 5.3.06 を参考にしている三訂版を経て 2002 年 6 月現在の最新バージョンであるバージョン 6.1.12 に基づく四訂版となっています.

Mizar は頻繁にバージョンアップされますのでご了承ください.

著者

参考文献

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

索引

/\, 13
∩, 13

ABSTR, 58
ABSTRact, 58
abstract file, 17
ACCOM, 59
accommodate, 59
and, 26
Any, 39
article, 3, 9
assume, 20, 27
attribute, 48

be, 41
begin, 9

c=, 41
case, 31, 32
cases, 31, 32
CHECKVOC, 44
CHKLAB, 62
cluster, 48, 50
consider, 29
contradiction, 22

def, 15
definition, 17
DICT, 57

end, 19
environ, 9
equals, 47
ERRFLAG, 60
ex, 29
existence, 46

FINDVOC, 10
for, 24
Formalized Mathematics, 64, 122
functor, 5, 45

given, 30

hence, 16, 19
hereby, 36
HIDDEN.ABS, 5
HIDDEN.VOC, 11, 43
holds, 24

iff, 6, 23
implies, 6
in, 13, 14
INACC, 62
IRRTHS, 63
IRRVOC, 63

let, 25
Linux 版, 7, 53, 55, 58, 120
LISTVOC, 10

MIZ2ABS, 64
MIZ2PREL, 64
Mizar Society, 3
Mizar 言語　3, 5
Mizar チェッカー, 59
Mizar プロジェクト, 3
MIZF, 60
mode, 39
MS-DOS 版, 7, 53, 54, 57, 58, 120

Nat, 39
not, 5
now, 23

or, 5

per, 32
predicate, 5, 44
PREL, 57, 58
proof, 19
proof ～ end, 16

Real, 39, 67
reconsider, 40
RELPREM, 61
reserve, 47

set, 13, 39, 40
st, 25
such that, 26

take, 29, 34
TARSKI, 5, 42
terminology, 67
TEXT, 57
then, 16
theorem, 17
thesis, 20
thus, 16, 19
TRIVDEMO, 62

uniqueness, 46

vocabulary file, 5, 10, 43
VERIFIER, 59

引用部, 15, 16

オブジェクト指向言語, 40

階層構造, 40
拡張文字, 53
仮定部, 20
環境設定, 3, 10
環境部, 9, 10

コメント, 18, 60
固有名詞, 30

述語論理, 5, 6
証明のスケルトン, 22
証明の飛躍, 14
証明部, 46, 48

全称記号, 6, 24

存在記号, 6, 29, 35

定義, 43

同値, 6, 23

場合分け 31
背理法, 21

本体部, 9, 15

優先順位, 44

予約語, 5, 73

ラベル, 15, 32, 33

論理式, 6

Mizar講義録

1994年3月16日　発行
1996年7月29日　改訂発行
1998年9月10日　三訂版
2002年10月1日　四訂版

著作者　中村八束，渡辺稔彦，田中保史，カワモト・ポーリン
発行者　信州大学工学部 情報基礎研究室

発行　〒380－8553 長野県長野市若里4-17-1
信州大学工学部情報工学科
中村研究室
TEL 026 (269) 5495