Mizar: An Impression

Freek Wiedijk
Nijmegen University
<freek@cs.kun.nl>

Abstract

This note presents an introduction to the Mizar system, followed by a brief comparison between Mizar and Coq. Appended are a Mizar grammar and an annotated example of a complete Mizar article.

1 What It Is

Mizar is a system for representing mathematical proof in a computer such that a program can check the correctness. It has been developed by Andrzej Trybulec and his team in Bialystok, Poland since about 1973. The Mizar language is quite close to the language of informal mathematics ('the mathematical vernacular'). Mizar is based on a ZF-like set theory with classical first order logic. Part of the Mizar project is the development of a large database of mathematics which currently numbers 587 'articles' taking 41 megabytes (even without the proofs it's 6 megabytes). A major project currently underway is the translation to Mizar (under direction of Grzegorz Bancerek) of a real mathematics book: 'A Compendium of Continuous Lattices' [1].

Mizar has had a number of dialects. 'Mizar MSE' (also called 'baby Mizar') is a toy language not meant for real applications. The current implementation of the full Mizar language is called 'PC Mizar'. It's written in Turbo/Borland Pascal and runs only under DOS/Windows. Functionally it's basically one program, '`mizf`', that non-interactively checks a Mizar file for its correctness.

The Mizar system doesn't have much documentation. The text that most resembles a manual[1] is 'An Outline of PC Mizar' by Michał Muzalewski [3].

A way to explore the Mizar language that works surprisingly well is to study the Mizar syntax (see appendix B on page 16 for a context free grammar) and for specific constructions to search the Mizar library for examples.

2 Definitions

A Mizar article consists of an '`environ`' header which indicates what other articles from the Mizar library it refers to, followed by a sequence of definitions and theorems.

A definition has the general shape:

```
definition
  let arguments;
  assume preconditions;
  func pattern -> type means :label: statement;
```

[1] An electronic version of this manual is available on the World Wide Web at the address <http://www.cs.kun.nl/~freek/mizar/mizarmanual.ps.gz>

```
  correctness proof
end;
```

The *pattern* presents the way the operation is written (both normal function notation and operator notation are possible; and not all arguments need to be present in the pattern: Mizar supports implicit arguments). In the defining *statement* the defined object is referred to as '`it`'. The correctness proof has to show existence and uniqueness of the defined object given the preconditions.

A definition of the form:

```
func pattern -> type means :label:
  it = expression;
```

can be abbreviated as:

```
func pattern -> type equals :label: expression;
```

Here is an example of a definition, the definition of '`log`' (from article '`POWER`'):

```
reserve a,b for Real;
definition
  let a,b;
  assume A1: a>0 & a<>1 and A2: b>0;
  func log(a,b) -> Real means
:Def3: a to_power it = b;
  existence
251 lines of existence proof omitted
  uniqueness by A1,Th57;
end;
```

(The '`reserve`' statement reserves variables for a certain type: the type of a reserved variable needn't be given. So because of the '`reserve`' statement in the first line, the '`let a,b`' means '`let a,b be Real`'.)

And here is the definition of subtraction (from '`REAL_1`'):

```
reserve x,y for Real;
definition
  let x,y;
  func x-y -> Real equals :Def3: x+(-y);
  correctness;
end;
```

(The '`correctness`' at the end is an abbreviation of all elements of the correctness proof that are left to be proved: for a definition with '`means`' these are '`existence`' and '`uniqueness`'; and for a definition with '`equals`' it's '`coherence`', which says that the object has the correct type. In this example correctness is obvious, so there's no '`by`' justification needed.)

Apart from '`func`' definitions for functions, Mizar has '`pred`' definitions for predicates and '`mode`' and '`attr`' definitions for types (for an explanation of Mizar types, see section 5 below). These are syntactically rather similar to the '`func`' definitions and will not be explained in detail.

In the definition of functions and predicates synonyms and antonyms can be given and properties like commutativity, symmetry, reflexivity and irreflexivity can be indicated. The Mizar system will magically 'know' about these things, and variations on the same expression will behave like they are merely syntactic variants. For instance because the definition of $\leq$ (in '`ARYTM`') is:

```
definition
  let x,y be Element of REAL;
  pred x ≤ y means
6 lines of definition
89 lines of correctness proof
  synonym y ≥ x;
  antonym y < x;
  antonym x > y;
end;
```

it doesn't matter whether one writes `x < y` or `y > x`. Also it means that theorems about `<` often also are usable (probably with a contraposition of an implication) to prove statements about $\geq$.

3 Theorems

The major part of a Mizar article consists of theorems (just like the major part of a computer program source consists of procedures/functions). A theorem has the shape:

```
theorem label: statement
proof
  proof steps
end;
```

An example of a theorem (from article ‘`IRRAT_1`’):

```
theorem T2:
  ex x, y st x is irrational & y is irrational &
    x.^.y is rational
proof
  set w = √2;
  H1: w is irrational by INT_2:44,T1;
  w>0 by AXIOMS:22,SQUARE_1:84;
  then (w.^.w).^.w = w.^.(w•w) by POWER:38
    .= w.^.(w²) by SQUARE_1:58
    .= w.^.2 by SQUARE_1:88
    .= w² by POWER:53
    .= 2 by SQUARE_1:88;
  then H2: (w.^.w).^.w is rational by RAT_1:8;
  per cases;
  suppose H3: w.^.w is rational;
    take w, w;
    thus thesis by H1,H3;
  suppose H4: w.^.w is irrational;
    take w.^.w, w;
    thus thesis by H1,H2,H4;
end;
```

(Note that Mizar uses ‘high ASCII’ characters, like ‘$\surd$’, ‘$\bullet$’ and ‘^{2}’ which in the DOS character set have ASCII codes 251, 249 and 253.) Various elements in this example will be discussed below.

Mizar statements are those of the language of first order logic (using the keywords ‘`contradiction`’, ‘`not`’, ‘`&`’, ‘`or`’, ‘`implies`’, ‘`iff`’, ‘`for ... holds`’ and ‘`ex`

... `st`'). The atomic formulas of this language either are instances of predicates (often written as an infix operator, with '`=`' being one of the most important), or statements stating that an expression '`is`' a type or adjective.

The only slight deviation from this is that a combination of a universal quantification and an implication is usually not written as:

`for` *variable* `holds (`*statement* `implies` *statement*`)`

but as:

`for` *variable* `st` *statement* `holds` *statement*

A source of confusion might be that Mizar has both '`&`' versus '`and`' and '`st`' versus '`such that`'. The first variants are syntax of first order formulas, the second variants are keywords that are part of Mizar statements.

Another 'duplication' is that Mizar has defined '`func`' functions (Mizar calls those defined operations 'functors'), versus functions in the underlying set theory (sets of Kuratowski pairs). An application of the first kind of function is written `f(x)` while that of the second kind is `f.x` (with an infix dot operator).

A third such 'duplication' is between '`x` ∈ `X`' and '`x is Element of X`'. The first is the primitive binary predicate from set theory, the second is a typing statement (saying that '`x`' has type '`Element of X`'). For each set, '`Element of`' gives the type of its elements. For instance: '`REAL`' is a set, but '`Real`' which is defined to be '`Element of REAL`' is a type. So it is '`x` ∈ `REAL`', but '`x is Real`' or '`x is Element of REAL`'.

A Mizar proof consists of a list of proof steps. Such a proof step basically has the shape:

label`:` *statement* `by` *labels*`;`

For instance, an example of a Mizar proof step is:

```
H1: w is irrational by INT_2:44,T1;
```

Here the statement is '`w is irrational`', it's labeled '`H1`', and it's a consequence of the statements labeled '`INT_2:44`' and '`T1`', which are:

```
:: INT_2:44
  2 is prime
```

(the 44th theorem from article '`INT_2`') and:

```
T1: p is prime implies sqrt p is irrational
```

(which was proved earlier in the same file).

A label in the list after the '`by`' keyword either is a label from an earlier theorem or proof step in the same file, or it's a reference to a theorem from a different Mizar article. In the latter case it looks like:

article name`:`*sequence number*

(so while 'local' references are by name, 'outside' references are by number). One can look up these sequence numbers by browsing the 'abstract' files of the library, which are the articles in which automatically proofs have been removed and sequence numbers have been inserted.

Instead of referring to the label of the immediately preceding step, one may also prefix the proof step with the keyword '`then`'. So:

```
A: statement by labels;
B: next statement by A, more labels;
```

can be written as:

```
statement by labels;
then B: next statement by more labels;
```

Apart from these kind of steps (called 'diffuse' reasoning steps) there are steps that relate to the statement that's being proved ('skeleton' steps). At each moment there is a 'current' statement that has to be proved: the skeleton steps transform that statement. For instance suppose that the statement to be proved looks like:

```
A implies B & C
```

Then the following skeleton proves it, using the '`assume`' and '`thus`' skeleton steps:

```
assume label: A;
diffuse steps
thus B by labels;
more diffuse steps
thus C by labels;
```

After the '`assume`' step the statement to be proved has been reduced to `B & C`. And after the first '`thus`' it has become `C`.

The combination '`then thus`' isn't syntactically correct: it is written as '`hence`'. So '`hence`' means that the statement follows from the previous step ('`then`') and that it is part of what was to be proved ('`thus`').

So at each moment in a Mizar proof there's a statement that's left to be proved. This statement may be referred to by the keyword '`thesis`'. Often a proof or subproof ends with '`hence thesis`' (the Mizar library contains this construction 36885 times).

The '`assume`' and '`thus`' steps correspond in the system of natural deduction to implication introduction and conjunction introduction. Other skeleton steps are:

- '`let`' for universal introduction:

  ```
  let variable be type;
  ```

- '`consider`' for existential elimination:

  ```
  consider variable being type such that properties
    by labels;
  ```

  (the statements referred to by the *labels* have to justify the appropriate existential statement)

- '`take`' for existential introduction:

  ```
  take expression;
  ```

- '`per cases`' for disjunction elimination:

  ```
  per cases by labels;
  suppose label: statement;
    proof for the first case
  suppose label: another statement;
    proof for the second case
  more cases
  ```

Not all natural deduction rules have a Mizar skeleton step for a counterpart: some are handled as a diffuse step.

In Mizar a proof can contain subproofs. The part of a proof step of the form:

```
... by labels;
```

is called its justification. It is not full first order provability, but some weaker variant that can be decided quickly (approximately: in only one of the premises universal quantors may be instantiated; on the other hand it's quite good at reasoning from type information, at applying equalities, and at deducing existential statements). This 'by' justification sometimes isn't sufficient: so a statement also may also be justified by a full proof in the form:

```
label: statement
proof
  proof steps
end;
```

Note that there is no semicolon after the *statement* (that semicolon would count for an 'empty' justification). In the case that a full proof like this is given, the statement can be omitted with the 'now ... end' construction:

```
label:
now
  proof steps
end;
```

In that case the statement that is proved (that the *label* refers to) is 'calculated' from the proof.

Mizar also has support for equational reasoning. One can write:

```
label: expression = expression by labels
  .= expression by labels
  more steps
  .= expression by labels;
```

The transitivity of these equalities is handled automatically: the *label* will refer to the equality of the initial and final expressions.

And Mizar supports higher order statements. When invoking such a 'scheme' its name is not put after the keyword 'by' but after the keyword 'from', and it takes arguments. These arguments are *not* the 'higher order parameters' which are between braces in the definition of the scheme (those are determined automatically), but instead the 'conditions' after the 'provided'.

The scheme for doing induction on natural numbers is defined (in article 'NAT_1') by:

```
scheme Ind { P[Nat] } :
  for k holds P[k]
provided
  A1: P[0] and
  A2: for k st P[k] holds P[k+1]
16 lines of proof omitted
```

(the square brackets mean that P is a predicate: functions are written with round brackets; even when such a parameter function is a constant those brackets have to be written). The 'Ind' scheme then is applied like:

```
label: statement from Ind(label,label);
```

in which the two labels refer to the instances of the two 'provided' statements that this scheme needs (which of course are the base and induction step cases).

4 Syntax

Expressions in a Mizar text look rather mathematical. This is caused by two things: the Mizar library uses the high ASCII part of the DOS character set (which contains lots of 'mathematical' symbols) and in Mizar-expressions various styles of operators are allowed (prefix, postfix, infix and 'bracket-like').

The Mizar lexical syntax isn't fixed. In fact, a Mizar article generally consists of two files: the article file (with a name ending with '`.miz`') and the vocabulary file (ending with '`.voc`'; which has to be in a subdirectory called '`dict`' in order for the system to be able to find it). This last kind of file presents lexical elements. It contains both the 'identifiers' from the definitions as well as the operator symbols. Such a vocabulary file contains for each lexical element a line that starts with a capital letter giving the lexical category of the symbol (the symbols for functions, predicates, modes, etc. all have *different* lexical categories), then the symbol itself and then possibly a priority.

So the Mizar library really consists of a number of articles together with a number of vocabularies. The articles can be browsed in source format, but the vocabularies are only present in 'compiled' form. The program '`findvoc -w`' is used to find out from which vocabulary a specific symbol comes. For instance the command:

```
findvoc -w .
```

has as output:

```
vocabulary: FUNC
O.  100
```

which tells us that '`.`' is an operator symbol (at the lexical level there's no difference between infix, prefix or postfix operators, so it can be used in all three ways) from the vocabulary '`FUNC`' and that it has priority 100 (if there are multiple definitions of a '`.`' operator they *all* have this priority).

For the analysis of the expression:

```
f.x
```

three aspects can be distinguished: the lexical analysis (there are three tokens: '`f`', '`.`' and '`x`'), the way the expression has to be parsed (the pattern from:

```
reserve x for set;
reserve f for Function;
definition
  let f,x;
  func f.x -> set means
:: FUNCT_1:def 4
  [x,it] ∈ f if x ∈ dom f otherwise it = 0;
end;
```

is the one that applies here) and the 'notion' that it refers to (the 'meaning' of the '`.`' operator). These three aspects correspond exactly to the '`vocabulary`', '`notation`' and '`constructors`' directives in the '`environ`' header of an article. So in order to be able to process this '`f.x`' expression correctly, the directives:

```
vocabulary FUNC;
notation FUNCT_1;
constructors FUNCT_1;
```

have to be present. Note that, although often the names of a vocabulary and the article it belongs to are identical, this need not be the case (here it's `FUNC` versus `FUNCT_1`): the 'name spaces' of article names and vocabulary names are separate.

Although it's quite hard to get the '`environ`' header of an article right, these three directives are the ones that are most difficult to understand. The '`theorems`', '`schemes`' and '`clusters`' directives are straightforward: in order to be able to use a theorem, scheme or cluster from an article, the name of that article has to be present in the appropriate directive.

The '`definitions`' directive is *only* about definitional expansion (what in type theory is called 'delta reduction'). This directive says that the Mizar system is allowed to 'unfold' all definitions from the articles that are listed. The theorems (with '`def`' in front of the number) that stem from definitions are *theorems*, and the relevant directive for them is the '`theorems`' directive. The '`definitions`' directive is used rarely.

The '`requirements`' directive currently has only one possible instance:

```
requirements ARYTM;
```

It means that the Mizar system will 'know' about natural numbers and about some identities and inequalities between them.

For instance the inequality

```
1<>0;
```

doesn't need any justification when the `ARYTM` requirement is present.

A Mizar operator can take more than two arguments, but in that case brackets have to be around them and commas in between. So legal postfix operators are:

```
x f
(x) f
(x,y) f
...
```

and for instance a legal infix operator is:

```
(x,y) f (z,v,w)
```

Note that the 'ordinary' function application notation fits this paradigm.

Function identifiers and operator symbols have type '`O`' in a vocabulary file. In Mizar also bracket-like notation is allowed: for this the vocabulary types '`K`' and '`L`' are present. Because '`<*`' has type '`K`' and '`*>`' has type '`L`' (both from vocabulary '`FINSEQ`'), the expression:

```
<* x *>
```

is a legal pattern (it denotes a one element finite sequence).

5 Types

Although the semantics of Mizar is untyped set theory (the axioms of Mizar are the ZF axioms plus a rather strong axiom – which implies the axiom of choice – about the existence of arbitrarily large unreachable cardinals), the language itself is typed. But the types are a property of the expressions of the language, not of the objects (the sets) which those expressions refer to, so the language is not based on 'type theory'. Types are used to disambiguate expressions (operators can be overloaded and are determined by the types of their arguments) and for reasoning.

Mizar typings are either written as:

variable `be` *type*

or as:

variable `being` *type*

(in a '`let`' one probably would use the first and in a '`for`' one would use the second: but both variants may be used everywhere).

Mizar types are built from 'modes' and 'attributes'. A Mizar type is an instance of a mode (which is a parametrised type), possibly prefixed with a number of adjectives (an adjective is an attribute or the negation of an attribute).

For instance, in the type:

```
non empty Subset of NAT
```

the mode '`Subset`' is applied to the expression '`NAT`' (which denotes the set of natural numbers) giving the type '`Subset of NAT`'. To this type then the adjective '`non empty`' has been added, which is the negation of the attribute '`empty`'.

Every Mizar type has an ancestor type. This leads to a tree-like type hierarchy of which the root type is '`Any`' (or its synonym '`set`'). At every node of this tree there is a 'Boolean algebra'-like structure given by the adjectives. The Mizar system knows how to 'widen' types following this structure.

In Mizar it's possible to give an expression some explicit type. For this there is the '`reconsider`' construction, which is a variant of the '`set`' statement. The way to locally name an expression in Mizar is:

`set` *variable* `=` *expression*`;`

After this, *variable* behaves like *expression* has been substituted for it everywhere. A '`reconsider`' is like that, only then the *expression* gets a different type. It looks like:

`reconsider` *variable* `=` *expression* `as` *type*
  `by` *labels*`;`

(Note that the '`consider`' and '`reconsider`' statements are not related: the first uses an existential statement to find a new object with some properties, the second '`casts`' an already known object given by an expression to some type.)

The type machinery of Mizar is rather well developed. In particular Mizar has so-called 'clusters' to be able to automatically deduce extra type information: clusters allow the Mizar system to manipulate sets of adjectives. There are three kind of cluster definitions: the first kind states that some combination of adjectives gives a non-empty type (this needs to be known to the system, because Mizar types all have to be provably non-empty), the second kind tells that some combination of adjectives implies other adjectives, and the third kind tells that an expression of some shape has some adjectives.

An example of the first kind of cluster (from article '`HIDDEN`') is:

```
definition
  cluster non empty set;
end;
```

(there's no proof because '`HIDDEN`' and '`TARSKI`' are the two 'special' articles which supply the axiomatics of Mizar). It states that the type '`non empty set`' is inhabited. If this cluster weren't known, something like:

```
let X be non empty set;
```

wouldn't be allowed, because in that case the system wouldn't know that '`non empty set`' is a properly non-empty type.

An example of the second kind of cluster statement (from '`BINTREE1`') is:

```
definition
  cluster binary -> finite-order Tree;
28 lines of proof omitted
end;
```

This says that every expression that has type '`binary Tree`' has the adjective '`finite-order`' as well (so it really has the more informative type '`binary finite-order Tree`').

A third kind of cluster (from '`ABIAN`'):

```
definition
  let i be even Integer;
  cluster i+1 -> odd;
5 lines of proof omitted
end;
```

This last kind of cluster is very powerful, because lots of properties of expressions can be derived automatically with it, but it sometimes significantly slows down the system.

Mizar has the '`redefine`' statement, which means that an operation can have multiple definitions in which all but the first one have a '`redefine`' keyword added. Those redefinitions may be underspecified (all other information is copied from the original definition), they may be defined on a smaller set of arguments, but they do have to be proved 'compatible' with the original definition.

This can be used to give the same operation multiple types. For instance because of (from '`NAT_1`'):

```
reserve k,n for Nat;
definition
  let n,k;
  redefine func n+k -> Nat;
18 lines of proof omitted
end;
```

the Mizar system knows that the sum of two natural numbers will be a natural number (because the 'redefined' version of the operation is syntactically selected when the arguments are natural numbers: for that the articles in the '`notation`' directive have to be in the proper order) even although the addition operation originally was defined for real numbers (and in that definition has type `Real`). But because of the '`redefine`', although the redefinition has type `Nat`, all existing theorems about the original operation of type `Real` still apply for it as well.

6 Semantics

The semantics of Mizar is fairly straightforward. Mizar is about ZF style set theory with first order logic.

However, there is the subtlety of undefined expressions. The semantics of Mizar 'solves' the problem of undefined expressions in a way that leads to some unexpected properties. Firstly it means that types are *not* only 'syntactic sugar' coding predicates (instead they 'mean' something on the level of the semantics). And secondly it means that the axiom of choice is provable from the way Mizar implements first order logic.

A Mizar '`func`' operation can have preconditions. The proof that the operation is well defined uses these preconditions. However, when *applying* such an operation there is no need to prove that the preconditions hold: they may be false. So, while for instance a precondition of division is that the denominator has to be unequal to zero, one is allowed to write expressions like `1/0`.

The semantics of Mizar doesn't tell anything about the result of applying an operation for which the preconditions fail. If the preconditions are true, the defining statement from the definition is guaranteed to hold; but when the preconditions are false, you don't know anything. So `1/0` is an unknown object. A different way of looking at this is that the function that maps Mizar expressions to their meaning isn't unique: on 'undefined' expressions it can take any value.

The one exception to this rule is that the *type* of an undefined expression still has to apply. So, although we don't know what `1/0` is, we *do* know that it has to be a real number (as the division operation has type '`Real`'). So the defining statement (which is a predicate on the object that's being defined) is not relevant when the preconditions fail, but the type is.

This difference in semantical treatment of unary predicates and types coding those predicates is clear from the following example. Let '`something`' be some (irrelevant) statement. Then write:

```
reserve X for set;

definition
  let X;
  assume A: contradiction;
  func choice(X) -> Element of X means
:Def_choice: something;
  correctness by A;
end;

definition
  let X;
  assume A: contradiction;
  func choice1(X) means
:Def_choice1: it is Element of X & something;
  correctness by A;
end;
```

(Because the precondition is '`contradiction`' of course these operations are well defined whenever the preconditions hold.) The '`choice`' and '`choice1`' functions seem minor variations of each other. But because of the way the semantics treats undefined expressions we can prove:

```
theorem AC:
  choice(X) is Element of X;
```

but we *can't* prove:

```
theorem AC1:
  choice1(X) is Element of X
proof
  thus thesis by Def_choice1;
::>           *4
end;
```

(The `*4` line is the error message that the '`mizf`' checker inserts in the file, number `4` meaning that the '`by`' inference doesn't hold. It is explained by the Mizar checker as '`This inference is not accepted`'.)

Note that theorem '`AC`' gives a 'uniform' axiom of choice. (The axiom of choice also follows from Mizar's set-theoretical axioms, but it clearly already is 'hard wired' in the way Mizar treats first order logic.)

The meaning of the type '`Element of X`' when `X` is an empty set, is some unknown but non-empty class (all Mizar types are non-empty). In Mizar the statement:

```
ex x st x is Element of ∅
```

(stating that there exists some '`Element of ∅`') is provable ('`consider x being Element of ∅; take x;`'). But the seemingly contradictory statement:

```
not ex x st x ∈ ∅
```

is provable as well, because it's a theorem of ZF. (This does *not* mean that the semantics of Mizar isn't sound, but just that '`Element of`' has a strange interpretation.) One would expect the definition of the '`Element`' mode (in '`HIDDEN`') to have precondition '`X is non empty`', but this is not the case.

In order to 'build' mathematical structures, Mizar also has '`struct`' types. They are defined like:

```
struct(ancestor struct) struct name (#
  field name -> type,
  field name -> type,
  more fields
  field name -> type
#)
```

(The '`(#`' and '`#)`' brackets may be replaced with high ASCII characters looking like '≪' and '≫': in fact this notation is the more common one.)

An object of such a `struct` type is then written like:

```
struct name (# value,value, ... value #)
```

and a field is selected from a `struct` by:

```
the field name of struct expression
```

As an example of a `struct`, here is the way topological spaces are introduced in Mizar (from articles '`STRUCT_0`' and '`PRE_TOPC`'):

```
struct 1-sorted (#
  carrier -> set
#);

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

definition
  let IT be TopStruct;
  attr IT is TopSpace-like means
    the carrier of IT ∈ the topology of IT &
```

```
    (for a being Subset-Family of the carrier of IT
      st a c= the topology of IT
      holds union a ∈ the topology of IT) &
    (for a,b being Subset of the carrier of IT st
      a ∈ the topology of IT & b ∈ the topology of IT
      holds a ∩ b ∈ the topology of IT);
end;

definition
  mode TopSpace is TopSpace-like TopStruct;
end;
```

The *specific* semantics of structs in terms of the underlying set theory is not really interesting: some ad hoc coding will do the job (choosing some set for the field names and interpreting the `struct` as a partial function from that set will work). Because of the way `struct` types widen, an object will also be of a `struct` type if it has *more* fields than is required by the definition. The adjective '`strict`' means that such extra fields should be absent.

7 Comparison To Coq

The Mizar system is quite dissimilar to systems from the LCF tradition like Coq. We here will compare Mizar specifically to Coq, but a similar comparison holds to other LCF-like systems like HOL, Nuprl, Isabelle, etc.

The most striking difference between Mizar and Coq is that Mizar is a batch checker which checks a whole file at a time (the '`@proof`' keyword gives some control in this respect by allowing one to suppress the checking of specific proofs), while Coq is an interactive system (this is similar to the difference in spirit between compiled and interpreted programming languages). One of the places that this shows is in the readability of the formalizations: a Coq file is just a long list of 'commands', so it has a linear structure and the commands aren't designed for readability, while a Mizar text has much more structure and is quite accessible without having to 'run' the file.

Another difference is that Mizar doesn't have 'proof objects': the system doesn't reduce its correctness to the correctness of a small 'kernel' that only knows about a few primitives. There also is a difference in the kind of logic in the two systems: Coq's kernel is based on 'type theory' which naturally has a constructive logic, while Mizar is very much a classical system.

The Mizar project has *more* automation than Coq and it has *less* automation. The step used most in Mizar is the '`by`' inference; in Coq the common tactics are more elementary than that. The '`by`' inference knows how to reason with types, it is able to use equalities and it's able to logically combine the information from many statements (the maximum: 22 statements in '`GENEALG1`'). While it's not full first order inference its power is close to the kind of reasoning steps a human would take (the Mizar system has the '`relinfer`' program to eliminate superfluous steps, and running it shows that '`by`' is more powerful than one tends to expect). So for the majority of the steps Mizar is more powerful than Coq. On the other hand, the Coq system isn't 'closed', it can be extended with arbitrarily powerful 'tactics'. These tactics, like '`Omega`' and '`Ring`', are able to solve involved domain specific tasks and don't have a Mizar counterpart. Also, the Coq system is able to 'reflect' on algorithms: it's able to state an algorithm *inside* the system, to prove that it is correct, and then to execute it, making it possible to extend the system 'from within'. This also is a reason that Coq is, in a sense, more powerful than Mizar.

Another difference between Mizar and Coq is that Coq mainly reasons backwards from the statement that is to be proved, while Mizar reasons forwards. Basically the Mizar 'skeleton' steps correspond to the Coq tactics and reduce the statement that's to be proved, while the 'diffuse' steps reason forward from already known statements. Most of the steps in a Mizar proof are diffuse steps.

A difference that's not so much a difference between the languages or systems, as well as a difference between the projects, is that the Mizar project has given a big priority to the development of a large and organized library. The Mizar system intentionally makes it hard to create projects of more than a few files (it is possible, but the system gets slow), stimulating Mizar users to submit their work for integration into the Mizar library. And the Mizar library is continuously being reorganized and changed by the Mizar group (the 'library committee'), turning it into a structured and consistent whole. This wouldn't be possible if they didn't 'own' the files.

Here's an example of a change that has been made to the Mizar library at some time. In the Mizar library the real numbers are constructed from the rational numbers as Dedekind cuts. But in the construction of the set '`REAL`' it 'cuts out' the copy of the rationals and 'glues in' the 'original' rationals (similarly the integers are glued inside the rationals and the natural numbers glued inside the integers). That way the natural numbers are a subset of the reals but also the natural numbers are the *natural* natural numbers, i.e., the finite ordinals. So the real number 0 is the natural number 0 is the empty set. This cutting-and-pasting of sets of numbers is a 'hack', but it also is quite nice. The complex numbers don't contain the real numbers in this fashion yet, but a change like this 'to put the real numbers inside the complex numbers' has been planned.

So the Mizar library is significantly more well developed than the Coq library. For instance it already contains a full construction of the real numbers (the Coq library only has the real numbers axiomatically) with lots of properties proved. (On the other hand, the Mizar library currently doesn't yet contain a definition of the rather elementary notion of 'polynomial' so it's not *that* rich yet.) Also Mizar is much more mathematical than Coq. Mizar is primarily about abstract mathematics while Coq has much more a focus on topics from computer science like proving functional programs correct.

Yet another difference: Coq is much more 'mainstream science' than Mizar. Mizar is an old-fashioned system that not many people know (if we take the fair assumption that every Mizar user has written an article for the Mizar library then there are 117 people in the world knowing Mizar), there is hardly any documentation about it and it doesn't run on the usual platform for this kind of system (which is a Sun running Unix). On the other hand Coq actively follows the theoretical developments from the type theory community, it's well known, has many users, is well documented and is available for all major platforms.

John Harrrison has written a 'Mizar mode' for HOL. To create a Mizar interface for a LCF style prover like HOL or Coq, one has to write a tactic that is able to do '`by`' inference (Harrison decided to implement full first order provability, which is more powerful but less efficient), and one has to write a parser for the Mizar syntax. Because the semantics and the type systems of Mizar and Coq are not the same, there are differences there to be bridged too. Having a system that combines the strengths of Mizar and Coq will be nice (it will be a 'Mizar' that has proof objects) but doing it by 'enhancing' Coq will be rather inelegant (there will be duplications of functionality) and inefficient. And really, from the point of view of the mathematically inclined user Mizar already contains everything that one needs.

References

[1] G. Gierz, K.H. Hofmann, K. Keimel, J.D. Lawson, M. Mislove, and D.S. Scott. *A Compendium of Continuous Lattices.* Springer-Verlag, Berlin, Heidelberg, New York, 1980.

[2] J. Kotowicz, B. Madras, and M. Korolkiewicz. Basic notation of universal algebra. *Journal of Formalized Mathematics*, 4, 1992.

[3] M. Muzalewski. *An Outline of PC Mizar.* Fondation Philippe le Hodey, Brussels, 1993.

A How To Run Mizar

Select three directories:

- *distdir*: to unzip the Mizar distribution in
- *mizardir*: for the installed Mizar system
- *workdir*: for your articles

The recommended choice for *mizardir* is '`C:\MIZAR`', although everything works fine if you put it elsewhere.

To install the Mizar system, get the zipped installation files from:

`<ftp://ftp.mizar.org/ver`*version*`/disks.zip/>`

(the '*version*' of the system described in this note is '`5-3.07`'). Unzip them all inside *distdir*. Then run the `install` command:

distdir`\install` *distdir*`\` *mizardir*

This installs everything apart from the sources of the Mizar library in '*mizardir*`\MML`'. These sources are really not very important (not even for reference to the library: the abstracts in '*mizardir*`\ABSTR`' are much more suitable for that). To unpack the sources of the Mizar library too, run the '`FM`*number*`.EXE`' files from *distdir* inside '*mizardir*`\MML`'.

Now add:

`set mizfiles=`*mizardir*

to the appropriate '`AUTOEXEC.BAT`' file, add *mizardir* to the DOS search `PATH`, and the installation will be complete.

To write an article create two subdirectories:

- *workdir*`\TEXT`
- *workdir*`\DICT`

Supposing the name of the article you'll be going to work on is '`FOO`', then create:

- *workdir*`\TEXT\FOO.MIZ`
- *workdir*`\DICT\FOO.VOC`

which are your article and vocabulary files.

To check the article for correctness, `cd` to *workdir* and run the command:

```
mizf text\foo
```

(the `mizf` checker will add the suffix '`.miz`' by itself). This will check the file, and will put the error messages *inside* the checked file (so it might be necessary to close the file in the editor before running the '`mizf`' command).

B Grammar

```
Mizar-Article =
  'environ'
    { 'vocabulary' File-Name-List ';' |
      ( 'notation' |
        'constructors' |
        'clusters' |
        'definitions' |
        'theorems' |
        'schemes' ) File-Name-List ';' |
      'requirements' File-Name-List ';' }
  ( 'begin' { Text-Item } ) { ... } .
Text-Item =
  'reserve' Identifier-List 'for' Type-Expression-List ';' |
  'definition'
    { Definition-Item }
    [ 'redefine' { Definition-Item } ]
    'end' ';' |
  Structure-Definition |
  'theorem' Proposition Justification ';' |
  [ 'scheme' ] Identifier
    '{' ( Identifier-List '[' Type-Expression-List ']' |
          Identifier-List '(' Type-Expression-List ')' '->' Type-Expression )
        { ',' ... } '}'
    ':' Formula-Expression
    [ 'provided' Proposition { 'and' ... } ]
    Justification ';' |
  Auxiliary-Item |
  'canceled' [ Numeral ] ';' .
Definition-Item =
  Assumption |
  Auxiliary-Item |
  Structure-Definition |
  'mode' M-Symbol [ 'of' Identifier-List ]
    ( [ '->' Type-Expression ] [ 'means' Definiens ] ';'
          Correctness-Conditions |
      'is' Type-Expression ';' )
    { 'synonym' M-Symbol [ 'of' Identifier-List ] ';' } |
  'func' Functor-Pattern [ '->' Type-Expression ]
    [ ( 'means' | 'equals' ) Definiens ] ';'
    Correctness-Conditions
    { 'commutativity' Justification ';' }
    { 'synonym' Functor-Pattern ';' } |
  'pred' Predicate-Pattern [ 'means' Definiens ] ';'
    Correctness-Conditions
    { 'symmetry' Justification ';' |
      'connectedness' Justification ';' |
      'reflexivity' Justification ';' |
      'irreflexivity' Justification ';' }
    { ( 'synonym' | 'antonym' ) Predicate-Pattern ';' } |
```

‘attr’ *Identifier* ‘is’ V-*Symbol* ‘means’ *Definiens* ‘;’
[*Correctness-Conditions*]
{ (‘synonym’ | ‘antonym’)
(*Identifier* ‘is’ V-*Symbol* | *Predicate-Pattern*) ‘;’ } |
‘canceled’ [*Numeral*] |
‘cluster’ *Adjective-Cluster* *Type-Expression* ‘;’ *Correctness-Conditions* |
‘cluster’ *Adjective-Cluster* ‘->’ *Adjective-Cluster* *Type-Expression* ‘;’
Correctness-Conditions |
‘cluster’ *Term-Expression* ‘->’ *Adjective-Cluster* ‘;’
Correctness-Conditions .

Structure-Definition =
‘struct’ [‘(’ *Type-Expression-List* ‘)’] G-*Symbol* [‘over’ *Identifier-List*]
‘(#’ (U-*Symbol* { ‘,’ … } ‘->’ *Type-Expression*) { ‘,’ … } ‘#)’ ‘;’ .

Definiens =
[‘:’ *Identifier* ‘:’] (*Formula-Expression* | *Term-Expression*) |
[‘:’ *Identifier* ‘:’]
((*Formula-Expression* | *Term-Expression*) ‘if’ *Formula-Expression*)
{ ‘,’ … }
[‘otherwise’ (*Formula-Expression* | *Term-Expression*)] .

Functor-Pattern =
[*Functor-Loci*] O-*Symbol* [*Functor-Loci*] |
K-*Symbol* *Identifier-List* L-*Symbol* .

Functor-Loci =
Identifier |
‘(’ *Identifier-List* ‘)’ .

Predicate-Pattern = [*Identifier-List*] R-*Symbol* [*Identifier-List*] .

Correctness-Conditions =
{ ‘existence’ *Justification* ‘;’ |
‘uniqueness’ *Justification* ‘;’ |
‘coherence’ *Justification* ‘;’ |
‘compatibility’ *Justification* ‘;’ |
‘consistency’ *Justification* ‘;’ }
[‘correctness’ *Justification* ‘;’] .

Justification =
Simple-Justification |
(‘proof’ | ‘@proof’) *Reasoning* ‘end’ .

Reasoning =
{ *Reasoning-Item* }
[‘per’ ‘cases’ *Simple-Justification* ‘;’
((‘case’ (*Proposition* | *Conditions*) ‘;’ { *Reasoning-Item* })
{ … } |
(‘suppose’ (*Proposition* | *Conditions*) ‘;’ { *Reasoning-Item* })
{ … })] .

Reasoning-Item =
Auxiliary-Item |
Assumption |
(‘thus’ | ‘hence’) *Statement* |
‘take’ (*Term-Expression* | *Identifier* ‘=’ *Term-Expression*) { ‘,’ … } ‘;’ .

Auxiliary-Item =
[‘then’] *Statement* |
‘set’ (*Identifier* ‘=’ *Term-Expression*) { ‘,’ … } ‘;’ |

‘deffunc’ Identifier ‘(’ [Type-Expression-List] ‘)’ ‘=’ Term-Expression |
‘defpred’ Identifier ‘[’ [Type-Expression-List] ‘]’ ‘means’
Formula-Expression .

Assumption =
(‘let’ | ‘given’) Qualified-Variables [‘such’ Conditions] ‘;’ |
‘assume’ (Proposition | Conditions) ‘;’ .

Statement =
[‘then’]
(Proposition Justification ‘;’ |
‘consider’ Qualified-Variables [‘such’ Conditions]
Simple-Justification ‘;’ |
‘reconsider’
(Identifier ‘=’ Term-Expression | Identifier) { ‘,’ ... }
‘as’ Type-Expression Simple-Justification ‘;’ |
Term-Expression ‘=’ Term-Expression Simple-Justification
‘.=’ (Term-Expression Simple-Justification) { ‘.=’ ... }) |
[Identifier ‘:’] ‘now’ Reasoning ‘end’ ‘;’ .

Simple-Justification =
[‘by’ Reference { ‘,’ ... }] |
‘from’ Identifier [‘(’ Reference { ‘,’ ... } ‘)’] .

Reference =
Identifier |
File-Name ‘:’ (Numeral | ‘def’ Numeral) { ‘,’ ... } .

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

Proposition = [Identifier ‘:’] Formula-Expression .

Formula-Expression =
‘(’ Formula-Expression ‘)’ |
[Term-Expression-List] R-Symbol [Term-Expression-List] |
Identifier [‘[’ Term-Expression-List ‘]’] |
Term-Expression ‘is’ { [‘non’] V-Symbol } |
Term-Expression ‘is’ Type-Expression |
Quantified-Formula-Expression |
Formula-Expression ‘&’ Formula-Expression |
Formula-Expression ‘or’ Formula-Expression |
Formula-Expression ‘implies’ Formula-Expression |
Formula-Expression ‘iff’ Formula-Expression |
‘not’ Formula-Expression |
‘contradiction’ |
‘thesis’ .

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

Qualified-Variables =
Identifier-List |
(Identifier-List (‘being’ | ‘be’) Type-Expression) { ‘,’ ... }
[‘,’ Identifier-List] .

Type-Expression =
‘(’ Type-Expression ‘)’ |
Adjective-Cluster M-Symbol [‘of’ Term-Expression-List] |
Adjective-Cluster G-Symbol [‘over’ Term-Expression-List] .

Adjective-Cluster = { [‘non’] V-*Symbol* } .

Term-Expression =
  ‘(’ *Term-Expression* ‘)’ |
  [*Arguments*] O-*Symbol* [*Arguments*] |
  K-*Symbol* *Term-Expression-List* L-*Symbol* |
  Identifier ‘(’ [*Term-Expression-List*] ‘)’ |
  G-*Symbol* ‘(#’ *Term-Expression-List* ‘#)’ |
  Identifier |
  ‘{’ *Term-Expression*
    [(‘where’ *Identifier-List* ‘is’ *Type-Expression*) { ‘,’ … }]
    ‘:’ *Formula-Expression* ‘}’ |
  Numeral |
  Term-Expression ‘qua’ *Type-Expression* |
  ‘the’ U-*Symbol* ‘of’ *Term-Expression* |
  ‘the’ U-*Symbol* |
  ‘$1’ | ‘$2’ | ‘$3’ | ‘$4’ | ‘$5’ | ‘$6’ | ‘$7’ | ‘$8’ |
  ‘it’ .

Arguments =
  Term-Expression |
  ‘(’ *Term-Expression-List* ‘)’ .

File-Name-List = *File-Name* { ‘,’ … } .

Identifier-List = *Identifier* { ‘,’ … } .

Type-Expression-List = *Type-Expression* { ‘,’ … } .

Term-Expression-List = *Term-Expression* { ‘,’ … } .

C Axioms

Here are the undefined notions and axioms of the Mizar system: everything in the Mizar library is defined and proved from just this. (This is the content of articles ‘HIDDEN’ and ‘TARSKI’. Axiom ‘TARSKI:8’ was omitted because it’s derivable from ‘TARSKI:def 5’.)

```
definition mode Any; synonym set; end;
reserve x,y,z,u for Any, N,M,X,Y,Z for set;
definition let x,y; pred x = y; reflexivity; symmetry; antonym x <> y; end;
definition let x,X; pred x ∈ X; antisymmetry; end;
definition let X; attr X is empty; end;
definition cluster empty set; cluster non empty set; end;
definition func ∅ -> empty set; end;
definition let X; mode Element of X; end;
definition let X; func bool X -> non empty set; end;
definition let X; mode Subset of X is Element of bool X; end;
definition let X be non empty set; cluster non empty Subset of X; end;
definition let X,Y; pred X c= Y; reflexivity; end;
definition let D be non empty set, X be non empty Subset of D;
  redefine mode Element of X -> Element of D;
end;

theorem (for x holds x ∈ X iff x ∈ Y) implies X = Y;
definition let y; func {y} -> set means x ∈ it iff x = y;
  let z; func {y,z} -> set means x ∈ it iff x = y or x = z;
  commutativity;
```

```
end;
definition let y; cluster {y} -> non empty;
  let z; cluster {y,z} -> non empty;
end;
definition let X,Y; redefine pred X c= Y means x ∈ X implies x ∈ Y; end;
definition let X;
  func union X -> set means x ∈ it iff ex Y st x ∈ Y & Y ∈ X;
end;
theorem X = bool Y iff for Z holds Z ∈ X iff Z c= Y;
theorem x ∈ X implies ex Y st Y ∈ X & not ex x st x ∈ X & x ∈ Y;
scheme Fraenkel {A()->set, P[Any,Any]}:
    ex X st for x holds x ∈ X iff ex y st y ∈ A() & P[y,x]
  provided
    for x,y,z st P[x,y] & P[x,z] holds y = z;
definition let x,y; func [x,y] equals {{x,y},{x}}; end;
definition let X,Y;
  pred X ≈ Y means ex Z st
    (for x st x ∈ X ex y st y ∈ Y & [x,y] ∈ Z) &
    (for y st y ∈ Y ex x st x ∈ X & [x,y] ∈ Z) &
    for x,y,z,u st [x,y] ∈ Z & [z,u] ∈ Z holds x = z iff y = u;
end;
:: Axiom der unerreichbaren Mengen
theorem ex M st N ∈ M &
  (for X,Y holds X ∈ M & Y c= X implies Y ∈ M) &
  (for X holds X ∈ M implies bool X ∈ M) &
  (for X holds X c= M implies X ≈ M or X ∈ M);
```

D Complete Example

Here is an example of a complete Mizar text: it's the article 'UNIALG_1' from the Mizar library, article number 303 by Jarosław Kotowicz, Beata Madras and Małgorzata Korolkiewicz [2]. It defines the type 'Universal_Algebra' (together with the notions of 'arity' and 'signature') which is the Mizar implementation of 'one-sorted algebras' from the theory of universal algebra.

We both give here the 'abstract' file 'UNIALG_1.ABS' as well as the full Mizar article 'UNIALG_1.MIZ'. Interspersed are explanations. Sometimes the explanations contain Mizar text from other articles: the main text of the 'UNIALG_1' article can be recognized because it is not indented and because it has line numbers in the left margin.

D.1 Abstract

In order to find out what's in an article from the Mizar library, one generally only looks at its 'abstract' file. This is generated automatically from the full article. In it all proofs have been removed and 'sequence numbers' like 'UNIALG_1:5' for the theorems and 'UNIALG_1:def 11' for the definitions have been inserted automatically.

```
:: Basic Notation of Universal Algebra
::  by Jaros{\l}aw Kotowicz, Beata Madras and Ma{\l}gorzata Korolkiewicz
::
:: Received December 29, 1992
:: Copyright (c) 1992 Association of Mizar Users

environ

 vocabulary UNIALG, PFUNC1, FINSEQ, FUNC_REL, FUNC, FINITER2, PBOOLE, UNIALG_D;
 notation ARYTM, NAT_1, STRUCT_0, TARSKI, RELAT_1, FUNCT_1, FINSEQ_1, FUNCOP_1,
      PARTFUN1, ZF_REFLE;
 clusters TARSKI, FINSEQ_1, RELSET_1, STRUCT_0, ARYTM, PARTFUN1, FUNCOP_1;
```

```
13  constructors FINSEQ_4, STRUCT_0, ZF_REFLE, FUNCOP_1, PARTFUN1;
14  requirements ARYTM;
```

This is the 'environ' header of the article. It mentions the various vocabularies (in the 'vocabulary' directive) and articles (in the 'notation', 'clusters' and 'constructors' directives) from the Mizar library that the article uses. The only special items in this header are the vocabulary 'UNIALG' which is the one that's 'special' to this article, and the 'ARYTM' label, which is neither a vocabulary name nor an article name.

The vocabulary file 'UNIALG.VOC' is not explicitely present in the Mizar distribution but only as part of the 'compiled' vocabulary library 'MML.VCB'. However, it can be printed with the command 'listvoc UNIALG':

```
GUAStr
Ucharact
Vhomogeneous
Vquasi_total
Vpartial
OOpers 128
MUniversal_Algebra
Oarity 128
Osignature
```

These are the 'func' ('O'), 'mode' ('M'), 'attr' ('V'), 'struct' ('G') and 'struct field' ('U') symbols which are 'new' to this article. These symbols correspond to the definitions from lines 62, 63, 75, 82, 105, /, 142, 150 and 161 of the abstract, and lines 153, 154, 176, 182, 257, /, 343, 351 and 395 of the full article. The operation 'Opers' does not appear in this article, but is defined in article 'UNIALG_2' (both articles share the vocabulary).

```
17  begin
```

A Mizar article consists of one or more 'sections', each of which starts with 'begin' (there is no corresponding 'end'). This article only has one section.

```
20  reserve A for set,
21          a for Element of A,
22          x,y for FinSequence of A,
23          h for PartFunc of A*,A ,
24          n,m for Nat,
25          z for set;
```

These reservations are straight-forward: note that some of them only can be used when there's an 'A' in scope.

The most interesting type from this list is that of 'h':

```
PartFunc of A*,A
```

The postfix operator '*' is defined in article 'FINSEQ_1' (the easiest way to determine this is to click on it in the web version[2] of the Mizar library) as:

```
definition let D be set;
  func D* -> set means x ∈ it iff x is FinSequence of D;
end;
```

(definitions as presented here are taken from the abstracts and so don't contain proofs), with the operator symbol '*' defined in vocabulary 'FINSEQ' as:

[2] <http://www.mizar.org/JFM/mmlident.html>

```
0*  128
```

while the mode 'FinSequence' comes from the same article:

```
definition let n;
  func Seg n -> set equals { k : 1 ≤ k & k ≤ n };
end;

definition let IT be Relation;
  attr IT is FinSequence-like means ex n st dom IT = Seg n;
end;

definition
  mode FinSequence is FinSequence-like Function;
end;

definition let D be set;
  mode FinSequence of D -> FinSequence means rng it c= D;
end;
```

The 'PartFunc' mode is defined in article 'PARTFUN1' as:

```
definition let X,Y;
  mode PartFunc of X,Y is Function-like Relation of X,Y;
end;
```

so 'PartFunc of A*,A' means 'partial function from A* to A'. The general way to name a mode is:

mode-name `of` *parameter*`,`*parameter*`,`...

which explains the slightly unnatural syntax.

```
27 definition let A;
28   let IT be PartFunc of A*,A;
29 attr IT is homogeneous means
30 :: UNIALG_1:def 1
31  for x,y st x ∈ dom IT & y ∈ dom IT holds len x = len y;
32 end;

34 definition let A;
35  let IT be PartFunc of A*,A;
36 attr IT is quasi_total means
37 :: UNIALG_1:def 2
38  for x,y st len x = len y & x ∈ dom IT holds y ∈ dom IT;
39 end;

41 definition let A be non empty set;
42 cluster homogeneous quasi_total non empty PartFunc of A*,A;
43 end;
```

This cluster states that the type:

```
homogeneous quasi_total non empty PartFunc of A*,A
```

is inhabited. It is needed (because Mizar types have to be non-empty) to be allowed to use the adjectives 'homogeneous', 'quasi_total' and 'non empty' (alone or in combination) with types of the form 'PartFunc of A*,A'. Without it for instance the types:

```
homogeneous quasi_total non empty PartFunc of A*,A

homogeneous non empty PartFunc of A*,A

homogeneous non empty
  PartFunc of (the carrier of U)*,the carrier of U
```

in lines 50, 149 and 165-166 of this abstract wouldn't be legal (in the proofs there appear 11 more types like this).

```
theorem :: UNIALG_1:1
h is non empty iff dom h <> ∅;

theorem :: UNIALG_1:2
for A being non empty set, a being Element of A
holds {<∅>A} -->a is homogeneous quasi_total non empty PartFunc of A*,A;
```

The operator `<∅>A` denotes the empty finite sequence with elements of type `A`. It is defined in article ‘`FINSEQ_1`’ as:

```
definition redefine func ∅; synonym <∅>; end;

definition let D be set;
  func <∅>(D) -> empty FinSequence of D equals <∅>;
end;
```

where the operator symbol ‘`<∅>`’ is from vocabulary ‘`FINSEQ`’:

```
O<∅> 254
```

In this, ‘`∅`’ is the name of the empty set as introduced in article ‘`HIDDEN`’:

```
definition
  func ∅ -> empty set;
end;
```

and vocabulary ‘`HIDDEN`’:

```
O∅ 128
```

(Because ‘`HIDDEN`’ is one of the two files presenting the axioms of the Mizar system there is no ‘`means`’ or ‘`equals`’ in this definition. Also, the ‘`HIDDEN`’ article and the ‘`HIDDEN`’ vocabulary are always present and so they don’t need to be listed in the ‘`environ`’ header of the article.)

The braces ‘`{` … `}`’ denote the one element set. It is defined in article ‘`TARSKI`’:

```
definition let y;
  func {y} -> set means x ∈ it iff x = y;
end;
```

The symbols ‘`{`’ and ‘`}`’ occur on their own in the Mizar syntax too, and are not in a vocabulary.

The infix operator ‘`-->`’ creates a constant function on a set. It is defined in article ‘`FUNCOP_1`’:

```
definition let A, a be set;
  func A --> a -> set equals [:A, {a}:];
end;
```

and vocabulary ‘`FINITER2`’:

```
O-->  16
```

Here `[:` … `:]` is the Cartesian product from article ‘`ZFMISC_1`’:

```
definition let X1,X2;
  func [: X1,X2 :] means
    z ∈ it iff ex x,y st x ∈ X1 & y ∈ X2 & z = [x,y];
end;
```

and ‘`[` … `]`’ is the Kuratowski pair from article ‘`TARSKI`’:

```
definition
  func [x,y] equals {{x,y},{x}};
end;
```

```
52 theorem :: UNIALG_1:3
53 for A being non empty set, a being Element of A
54  holds {<∅>A} -->a is Element of PFuncs(A*,A);
```

The func ‘PFuncs’ is defined in article ‘PARTFUN1’:

```
definition let X,Y;
  func PFuncs(X,Y) -> set means
    x ∈ it iff
      ex f being Function st x = f & dom f c= X & rng f c= Y;
end;
```

```
56 definition let A;
57  mode PFuncFinSequence of A -> FinSequence of PFuncs(A*,A) means
58 :: UNIALG_1:def 3
59  not contradiction;
60 end;
```

Despite what the official Mizar grammar (appendix B on page 16) suggests, the redundant characterization ‘`means not contradiction`’ may not be omitted.

```
62 struct (1-sorted) UAStr ⟪ carrier -> set,
63       charact -> PFuncFinSequence of the carrier⟫;
```

So the `struct` ‘`UAStr`’ that underlies the implementation of the notion of ‘algebra’ in Mizar has two fields: ‘`carrier`’ which is the ‘sort’ of the algebra, and ‘`charact`’ which is the sequence of ‘functions’ of the algebra.

The ancestor `struct` ‘`1-sorted`’ is defined in article ‘`STRUCT_0`’:

```
definition
  struct 1-sorted ⟪carrier -> set⟫;
end;
```

```
65 definition
66  cluster non empty strict UAStr;
67 end;
```

This cluster states that there exists a `non empty strict UAStr`. It is not used in this article.

Note that because the type ‘`UAStr`’ widens to ‘`1-sorted`’, which is narrower than ‘`set`’, the adjective ‘`non empty`’ refers to the definition:

```
definition let S be 1-sorted;
  attr S is empty means the carrier of S is empty;
end;
```

from article ‘`STRUCT_0`’, instead of to the definition:

```
definition let X be set;
  attr X is empty;
end;
```

from article ‘`HIDDEN`’.

```
69 definition let D be non empty set, c be PFuncFinSequence of D;
70  cluster UAStr ⟪D,c ⟫ -> non empty;
71 end;
```

This cluster causes expressions of the shape ‘`UAStr⟪D,c⟫`’, with `D` having the adjective ‘`non empty`’, to gain the adjective ‘`non empty`’ too. It is used in line 323 of the article.

```
73 definition let A;
74  let IT be PFuncFinSequence of A;
75 attr IT is homogeneous means
76 :: UNIALG_1:def 4
77    for n,h st n ∈ dom IT & h = IT.n holds h is homogeneous;
78 end;

80 definition let A;
81  let IT be PFuncFinSequence of A;
82 attr IT is quasi_total means
83 :: UNIALG_1:def 5
84    for n,h st n ∈ dom IT & h = IT.n holds h is quasi_total;
85 end;

87 definition let F be Function;
88 redefine attr F is non-empty means
89 :: UNIALG_1:def 6
90  for n being set st n ∈ dom F holds F.n is non empty;
91 end;
```

The `attr` '`non-empty`' was defined in article '`ZF_REFLE`' as:

```
definition let F be Function;
  attr F is non-empty means not ∅ ∈ rng F;
end;
```

The redefinition that's given here has to be equivalent to this (it is allowed to have a more specific parameter type though, although here that's not the case).

Both the original definition and the redefinition are used in exactly the same way. This new definition is used in lines 242–250 of the full article (because the proof there is of the 'expanded' form of the definition: Mizar expands definitions from the article itself, and from articles in the '`definitions`' `environ` directive). Also its 'definitional theorem' ('`Def6`') is referred to in lines 301, 413 and 452 of the full article.

```
93 definition let A be non empty set; let x be Element of PFuncs(A*,A);
94  redefine
95   func <*x*> -> PFuncFinSequence of A;
96 end;
```

This redefines the '`<* ... *>`' operator from article '`FINSEQ_1`':

```
definition let x;
  func <*x*> -> set equals { [1,x] };
end;
```

This operator is used to write sequences of length one. The redefinition doesn't give a new characterization of this `func` (so it 'inherits' the original one), but it does change its *type*. Without it, adjectives like '`homogeneous`' and '`quasi_total`' wouldn't be applicable to expressions of the form '`<*x*>`', like for instance in line 123 of the abstract.

```
98 definition let A be non empty set;
99 cluster homogeneous quasi_total non-empty PFuncFinSequence of A;
100 end;
```

This cluster is not used in this article.

```
102 reserve U for UAStr;

104 definition let IT be UAStr;
105 attr IT is partial means
106 :: UNIALG_1:def 7
```

```
107   the charact of IT is homogeneous;
108  attr IT is quasi_total means
109  :: UNIALG_1:def 8
110   the charact of IT is quasi_total;
111  attr IT is non-empty means
112  :: UNIALG_1:def 9
113   the charact of IT <> <∅> & the charact of IT is non-empty;
114  end;

116  reserve A for non empty set,
117          h for PartFunc of A*,A  ,
118          x,y for FinSequence of A,
119          a for Element of A;
```

These reservations are identical to those from lines 20–23, apart that after this, '`A`' has the adjective '`non empty`'.

```
121  theorem :: UNIALG_1:4
122  for x be Element of PFuncs(A*,A) st x = {<∅>A} --> a holds
123    <*x*> is homogeneous quasi_total non-empty;

125  definition
126   cluster quasi_total partial non-empty strict non empty UAStr;
127  end;
```

This cluster is used five times in this article to establish the correctness of a type. For example it's used in the definition of the `mode` '`Universal_Algebra`' in line 142 of this abstract.

```
129  definition let U be partial UAStr;
130   cluster the charact of U -> homogeneous;
131  end;

133  definition let U be quasi_total UAStr;
134   cluster the charact of U -> quasi_total;
135  end;

137  definition let U be non-empty UAStr;
138   cluster the charact of U -> non-empty non empty;
139  end;
```

The '`non empty`' means that there is at least one function in an algebra, the '`non-empty`' means that these functions all are not empty.

```
141  definition
142  mode Universal_Algebra is quasi_total partial non-empty non empty UAStr;
143  end;

145  reserve U for partial non-empty non empty UAStr;

147  definition
148    let A;
149    let f be homogeneous non empty PartFunc of A*,A;
150  func arity(f) -> Nat means
151  :: UNIALG_1:def 10
152   x ∈ dom f implies it = len x;
153  end;
```

Note that the argument '`A`' of this `func` is an implicit argument: it's not present in the `notation` '`arity f`'.

```
155  theorem :: UNIALG_1:5
156  for U holds for n st n∈dom the charact of(U) holds
157    (the charact of(U)).n is
158     PartFunc of (the carrier of U)*,the carrier of U;
```

```
definition let U;
func signature(U) ->FinSequence of NAT means
:: UNIALG_1:def 11
 len it = len the charact of(U) &
 for n st n ∈ dom it holds
   for h be homogeneous non empty
   PartFunc of (the carrier of U )*,the carrier of U
     st h = (the charact of(U)).n
    holds it.n = arity(h);
end;
```

D.2 Article

```
1  :: Basic Notation of Universal Algebra
2  ::  by Jaros{\l}aw Kotowicz, Beata Madras and Ma{\l}gorzata Korolkiewicz
3  ::
4  :: Received December 29, 1992
5  :: Copyright (c) 1992 Association of Mizar Users

7  environron

9   vocabulary UNIALG, PFUNC1, FINSEQ, FUNC_REL, FUNC, FINITER2, PBOOLE, UNIALG_D;
10  constructors FINSEQ_4, STRUCT_0, ZF_REFLE, FUNCOP_1, PARTFUN1;
11  requirements ARYTM;
12  notation ARYTM, NAT_1, STRUCT_0, TARSKI, RELAT_1, FUNCT_1, FINSEQ_1, FUNCOP_1,
13       PARTFUN1, ZF_REFLE;
14  clusters TARSKI, FINSEQ_1, RELSET_1, STRUCT_0, ARYTM, PARTFUN1, FUNCOP_1;
15  definitions TARSKI, STRUCT_0, ZF_REFLE;
16  theorems TARSKI, FUNCT_1, PARTFUN1, FINSEQ_1, FUNCOP_1, RELAT_1, RELSET_1,
17       FINSEQ_3, ZF_REFLE;
18  schemes MATRIX_2;
```

The '`definitions`', '`theorems`' and '`schemes`' directives were not in the abstract because they are only relevant to the proofs. They also contain names of articles from the library. The last two list articles from which `theorems` and `schemes` are used.

The '`definitions`' directive from line 15 applies to 9 places in the article:

- The proofs in lines 50–54, 56–60, 93–97, 99–103, 131–135 and 137–141. These proofs are of statements involving the inclusion operator '`c=`', but the proofs prove a universally quantified formula. The '`c=`' operator is defined in article '`TARSKI`' (*re*defined really, because the '`c=`' operator is originally introduced without definition in article '`HIDDEN`') as:

  ```
  definition let X,Y;
    redefine pred X c= Y means x ∈ X implies x ∈ Y;
  end;
  ```

  and so with a '`definitions TARSKI;`' directive the '`c=`' statement will be expanded to a suitable universal formula.

- The '`thus the carrier of UAStr⟨⟨D,c⟩⟩ is non empty;`' in lines 161 and 170. Here the statement to be proved is about a `struct`, but the statement that is supplied is about `the carrier of` that `struct`. In order for this to be correct a '`definitions STRUCT_0;`' has to make the definition of '`empty`' on structs transparent:

  ```
  definition let S be 1-sorted;
    attr S is empty means the carrier of S is empty;
  end;
  ```

- The proof '`assume ∅ ∈ rng F; ... hence contradiction by A2;`' in lines 199–201. It proves '`not ∅ ∈ rng F`', but the statement it is supposed to prove really is '`F is non-empty`'. So the directive '`definitions ZF_REFLE;`' is needed to make the definition:

  ```
  definition let F be Function;
    attr F is non-empty means not ∅ ∈ rng F;
  end;
  ```

  transparent.

It is '`Analyzer`' (the type checking phase of the Mizar checker) that needs these definitional expansions, instead of '`Checker`' (which checks the logical correctness of the inferences).

```
20 begin

23 reserve A for set,
24         a for Element of A,
25         x,y for FinSequence of A,
26         h for PartFunc of A*,A ,
27         n,m for Nat,
28         z for set;

30 definition let A;
31   let IT be PartFunc of A*,A;
32 attr IT is homogeneous means                 :Def1:
33  for x,y st x ∈ dom IT & y ∈ dom IT holds len x = len y;
34 end;
```

Note that this `definition` is referred to (because of the label on line 32) as '`Def1`' in this article (on line 364), but (the label on line 30 of the abstract) as '`UNIALG_1:def 1`' in the other articles (specifically: articles '`ALG_1`', '`FREEALG`' and '`PRALG_1`'). Similarly the `theorem` from lines 83–84 is referred to as '`Th1`' in this article (lines 121, 357 and 372), but as '`UNIALG_1:1`' (line 45 of the abstract) in the others ('`ALG_1`', '`MSSUBLAT`', '`PRALG_1`' and '`UNIALG_2`').

```
36 definition let A;
37  let IT be PartFunc of A*,A;
38 attr IT is quasi_total means
39  for x,y st len x = len y & x ∈ dom IT holds y ∈ dom IT;
40 end;
```

The following definition and its proof will be annotated in more detail than the remainder of the Mizar text. The proof (lines 45–80) is the really same as the proof of `theorem` '`Th2`' (lines 89–122 on page 35), so it can be read there without interruptions.

```
42 definition let A be non empty set;
43 cluster homogeneous quasi_total non empty PartFunc of A*,A;
```

This cluster states that the type:

```
homogeneous quasi_total non empty PartFunc of A*,A
```

is inhabited. It should be present if one wants to use the adjectives '`homogeneous`', '`quasi_total`' or '`non empty`' with types of the form '`PartFunc of A*,A`'.

```
44  existence
```

Because a '`cluster`' of this kind states the non-emptyness of a type, its correctness condition is an '`existence`' statement. This statement is:

```
ex f being PartFunc of A*,A st
  f is homogeneous & f is quasi_total & f is non empty
```

```
45  proof
46   consider a be Element of A;
```

A '`consider`' step needs an existential statement as a justification. In this case it is:

```
ex a be Element of A
```

(this isn't a full Mizar formula: append '`st not contradiction`' to complete it). Because Mizar types all are non-empty an existential statement of this kind is evident: so the '`consider`' needs no justification.

The property of '`A`' that (from line 42):

```
A is non empty
```

isn't used here: the '`consider`' step also would have been valid if '`A`' had just been a '`set`'. However, in that case:

```
a ∈ A
```

would not have been true (it is built into the Mizar system that when '`A is non empty`' it follows from '`a is Element of A`' that '`a ∈ A`'), and then the '`hence`' step in line 59 would have failed.

```
47   set f = {<∅>A} -->a;
```

This is a local definition of a constant '`f`': all occurrences of '`f`' after this will be expanded to '`{<∅>A}-->a`'.

The '`f`' that's defined here is a function that has as domain the set '`{<∅>A}`' which it maps to the constant '`a`'. This domain consist of just the sequence of zero length: it is the set $\mathtt{A}^0$. So we have that:

$$\mathtt{f} : \mathtt{A}^0 \to \mathtt{A}, \quad \langle\rangle \mapsto \mathtt{a}$$

and so this `f` is the 'nullary' function on `A` which represents the constant `a`.

```
48   A1: dom f = {<∅>A} & rng f = {a} by FUNCOP_1:14;
```

This statement expands to:

```
A1: dom ({<∅>A}-->a) = {<∅>A} & rng ({<∅>A}-->a) = {a}
```

which follows from the type of '`A`' from line 42:

```
A is non empty set
```

the type of '`∅`' from article '`HIDDEN`':

```
definition
  func ∅ -> empty set;
end;
```

and:

```
theorem :: FUNCOP_1:14
  A <> ∅ implies dom (A --> x) = A & rng (A --> x) = {x};
```

```
49   A2: dom f c= A*
```

Because of the '`definitions TARSKI;`' directive from line 15, this expands according to:

```
definition let X,Y; redefine pred X c= Y means
:: TARSKI:def 3
  x ∈ X implies x ∈ Y;
end;
```

to:

```
A2: for z being set st z ∈ dom f holds z ∈ A*
```

```
50   proof
```

Here starts a 'local' subproof of the statement '`A2`'.

```
51     let z; assume z ∈ dom f; then
```

The '`let`' and '`assume`' skeleton steps correspond to the '`for` ...' and '`st` ... `holds`' parts of the statement. After these steps:

```
z ∈ A*
```

is left to be proved.

```
52     z = <∅>A by TARSKI:def 1,A1;
```

The statement:

```
z = <∅>A
```

follows from (because of the '`then`' at the end of line 51):

```
z ∈ dom f
```

and:

```
A1: dom f = {<∅>A} & rng f = {a}
```

and:

```
definition let y; func {y} -> set means
:: TARSKI:def 1
  x ∈ it iff x = y;
end
```

```
53     hence thesis by FINSEQ_1:65;
```

The statement that is left to be proved:

```
z ∈ A*
```

now follows from (the '`hence`' includes a '`then`'):

```
z = <∅>A
```

and the type of `<∅>A` given by:

```
definition let D be set;
  func <∅>(D) -> empty FinSequence of D equals
  :: FINSEQ_1:def 6
    <∅>;
end;
```

and:

```
theorem :: FINSEQ_1:65
  x ∈ D* iff x is FinSequence of D;
```

```
54    end;
```

... and that completes the subproof.

```
55  rng f c= A
56   proof
57    let z; assume z ∈ rng f; then
58    z = a by A1,TARSKI:def 1;
59    hence thesis;
60   end; then
```

This subproof is similar to the previous one. The step in line 59 uses that:

```
a ∈ A
```

This was explained in the text following the '`consider`' in line 46.

```
61  reconsider f as PartFunc of A*,A by RELSET_1:11,A2;
```

After this `reconsider` '`f`' will keep the same meaning (it will still expand to '`{<∅>A}-->a`'), but its *type* will have become '`PartFunc of A*,A`'.

In order to justify this, one has to prove:

```
f is PartFunc of A*,A
```

which because of the definition of '`f`' and the definition of `mode` '`PartFunc`':

```
definition let X,Y;
  mode PartFunc of X,Y is Function-like Relation of X,Y;
end;
```

'expands' to:

```
{<∅>A}-->a is Function-like Relation of A*,A
```

This follows from:

```
A2: dom f c= A*
```

and ('`then`' on line 60):

```
rng f c= A
```

and (from article '`FUNCOP_1`'):

```
definition let A, z be set;
  cluster A --> z -> Function-like Relation-like;
end;
```

and (from article '`RELAT_1`'):

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

and:

```
theorem :: RELSET_1:11
  for R being Relation st dom R c= X & rng R c= Y holds
    R is Relation of X,Y;
```

```
62  A3: f is homogeneous
```

This statement expands, because of the definition in lines 30–34, to:

```
A3: for x,y being FinSequence of A st x ∈ dom f & y ∈ dom f holds
  len x = len y
```

```
63    proof
64     let x,y be FinSequence of A; assume
65     x ∈ dom f & y ∈ dom f; then
```

After this, the statement left to be proved is:

```
len x = len y
```

```
66     x = <∅>A & y = <∅>A by TARSKI:def 1,A1;
```

The statement:

```
x = <∅>A & y = <∅>A
```

follows from (‘`then`’ in line 65):

```
x ∈ dom f & y ∈ dom f
```

and:

```
A1: dom f = {<∅>A} & rng f = {a}
```

and:

```
definition let y; func {y} -> set means
:: TARSKI:def 1
  x ∈ it iff x = y;
end
```

```
67     hence thesis;
```

Because (‘`hence`’ refers to the previous statement):

```
x = <∅>A & y = <∅>A
```

the `thesis` left to be proved:

```
len x = len y
```

is equivalent to:

```
len <∅>A = len <∅>A
```

which is true by reflexivity (equational reasoning is built into the Mizar system).

```
68    end;
69   A4: f is quasi_total
```

This statement expands, because of the definition in lines 36–40, to:

```
for x,y being FinSequence of A st len x = len y & x ∈ dom f holds
  y ∈ dom f
```

```
70    proof
71     let x,y be FinSequence of A; assume
72     A5: len x = len y & x ∈ dom f; then
```

After this, the statement left to be proved is:

```
y ∈ dom f
```

```
73     x = <∅>A by TARSKI:def 1,A1; then
```

The statement:

```
x = <∅>A
```

follows from:

```
A1: dom f = {<∅>A} & rng f = {a}
```

and (‘`then`’ in line 72):

```
len x = len y & x ∈ dom f
```

and:

```
definition let y; func {y} -> set means
:: TARSKI:def 1
  x ∈ it iff x = y;
end
```

```
74     len x = 0 by FINSEQ_1:32; then
```

The statement:

```
len x = 0
```

follows from (‘`then`’ in line 73):

```
x = <∅>A
```

and:

```
theorem :: FINSEQ_1:32
  p=<∅>(D) iff len p = 0;
```

```
75     y = <∅>A by FINSEQ_1:32,A5;
```

The statement:

```
y = <∅>A
```

follows from:

```
A5: len x = len y & x ∈ dom f
```

and (‘`then`’ in line 74):

```
len x = 0
```

and:

```
theorem :: FINSEQ_1:32
  p=<∅>(D) iff len p = 0;
```

```
76     hence thesis by A1,TARSKI:def 1;
```

The `thesis` left to be proved:

```
y ∈ dom f
```

follows from (‘`hence`’):

```
y = <∅>A
```

and:

```
A1: dom f = {<∅>A} & rng f = {a}
```

and:

```
definition let y; func {y} -> set means
:: TARSKI:def 1
  x ∈ it iff x = y;
end
```

```
77     end;
78   f is non empty by RELAT_1:60,FUNCOP_1:14;
```

The statement:

```
f is non empty
```

expands to:

```
{<∅>A}-->a is non empty
```

which follows from:

```
theorem :: RELAT_1:60
  dom ∅ = ∅ & rng ∅ = ∅;
```

and:

```
theorem :: FUNCOP_1:14
  A <> ∅ implies dom (A --> x) = A & rng (A --> x) = {x};
```

and the cluster (from article '`TARSKI`'):

```
definition let y;
  cluster {y} -> non empty;
end;
```

The knowledge that:

```
A is empty implies A = ∅
```

is built into the Mizar reasoner.

```
79   hence thesis by A3,A4;
```

The `thesis` to be proved is:

```
ex f being PartFunc of A*,A st
  f is homogeneous & f is quasi_total & f is non empty
```

This follows from the fact that for the specific 'defined' `f` from lines 47 and 61 we know that:

```
f is homogeneous & f is quasi_total & f is non empty
```

which follows from:

```
A3: f is homogeneous
```

and:

```
A4: f is quasi_total
```

and ('`hence`'):

```
      f is non empty
```

Note that Mizar is able to figure out the introduction of the existential quantifier `ex` by itself, without having to be told that '`f`' is the witnessing object.

```
80   end;
```

This finishes the annotated proof.

```
81  end;

83  theorem Th1:
84  h is non empty iff dom h <> ∅ by RELAT_1:64,RELAT_1:60;

86  theorem Th2:
87  for A being non empty set, a being Element of A
88  holds {<∅>A} -->a is homogeneous quasi_total non empty PartFunc of A*,A
```

Note that this theorem already has been proved inside the proof of the `cluster` (lines 47, 61, 62, 69 and 78), but that the `cluster` has to be present in order to state the theorem.

```
89   proof let A be non empty set, a be Element of A;
90     set f = {<∅>A} -->a;
91     A1: dom f = {<∅>A} & rng f = {a} by FUNCOP_1:14;
92     A2: dom f c= A*
93      proof
94       let z; assume z ∈ dom f; then
95       z = <∅>A by TARSKI:def 1,A1;
96       hence thesis by FINSEQ_1:65;
97      end;
98    rng f c= A
99     proof
100     let z; assume z ∈ rng f; then
101     z = a by A1,TARSKI:def 1;
102     hence thesis;
103    end; then
104   reconsider f as PartFunc of A*,A by RELSET_1:11,A2;
105   A3: f is homogeneous
106    proof
107     let x,y be FinSequence of A; assume
108     x ∈ dom f & y ∈ dom f; then
109     x = <∅>A & y = <∅>A by TARSKI:def 1,A1;
110     hence thesis;
111    end;
112   A4: f is quasi_total
113    proof
114     let x,y be FinSequence of A; assume
115     A5: len x = len y & x ∈ dom f; then
116     x = <∅>A by TARSKI:def 1,A1; then
117     len x = 0 by FINSEQ_1:32; then
118     y = <∅>A by FINSEQ_1:32,A5;
119     hence thesis by A1,TARSKI:def 1;
120    end;
121   thus thesis by A3,A4,A1,Th1;
122  end;

124 theorem Th3:
125 for A being non empty set, a being Element of A
126  holds {<∅>A} -->a is Element of PFuncs(A*,A)
127  proof let A be non empty set, a be Element of A;
128    set f = {<∅>A} -->a;
129    A1: dom f = {<∅>A} & rng f = {a} by FUNCOP_1:14;
130    A2: dom f c= A*
131     proof
132      let z; assume z ∈ dom f; then
133      z = <∅>A by TARSKI:def 1,A1;
```

```
134      hence thesis by FINSEQ_1:65;
135     end;
136   rng f c= A
137    proof
138     let z; assume z ∈ rng f; then
139     z = a by A1,TARSKI:def 1;
140     hence thesis;
141    end; then
142   reconsider f as PartFunc of A*,A by RELSET_1:11,A2;
143   f ∈ PFuncs(A*,A) by PARTFUN1:119;
144   hence {<∅>A} -->a is Element of PFuncs(A*,A);
145  end;

147 definition let A;
148  mode PFuncFinSequence of A -> FinSequence of PFuncs(A*,A) means
149 :Def3: not contradiction;
150  existence;
```

The correctness condition of a `mode` definition is the non-emptiness of the defined type, which is an '`existence`' statement. In this case this statement becomes:

```
ex c being FinSequence of PFuncs(A*,A) st not contradiction
```

Because Mizar types all are non-empty, the existence of a '`c`' of the appropriate type is obvious, and of course it satisfies '`not contradiction`'. So this statement needs no justification.

```
151 end;

153 struct (1-sorted) UAStr ⟪ carrier -> set,
154       charact -> PFuncFinSequence of the carrier⟫;

156 definition
157  cluster non empty strict UAStr;
158  existence
159  proof consider D being non empty set, c being PFuncFinSequence of D;
160   take UAStr ⟪D,c ⟫;
161   thus the carrier of UAStr ⟪D,c ⟫ is non empty;
162   thus thesis;
163  end;
164 end;

166 definition let D be non empty set, c be PFuncFinSequence of D;
167  cluster UAStr ⟪D,c ⟫ -> non empty;
168  coherence
```

The `correctness` condition of a `cluster` of this kind is called '`coherence`'. Here, of course, it is:

```
UAStr⟪D,c⟫ is non empty
```

Because of the '`definitions STRUCT_0;`' directive in line 15 this expands to:

```
the carrier of UAStr⟪D,c⟫ is non empty
```

which reduces to:

```
D is non empty
```

which follows from the type of '`D`' in line 166.

```
169  proof
170   thus the carrier of UAStr ⟪D,c ⟫ is non empty;
171  end;
172 end;

174 definition let A;
```

```
 let IT be PFuncFinSequence of A;
attr IT is homogeneous means      :Def4:
   for n,h st n ∈ dom IT & h = IT.n holds h is homogeneous;
end;

definition let A;
 let IT be PFuncFinSequence of A;
attr IT is quasi_total means     :Def5:
   for n,h st n ∈ dom IT & h = IT.n holds h is quasi_total;
end;

definition let F be Function;
redefine attr F is non-empty means       :Def6:
 for n being set st n ∈ dom F holds F.n is non empty;
 compatibility
```

The correctness condition for the redefinition of a 'pred' or 'attr' is called 'compatibility'. In this case, it is:

```
F is non empty iff
  for n being set st n ∈ dom F holds F.n is non empty
```

In this statement and its proof 'non empty' still has the 'old' definition, which is:

```
definition let F be Function;
  attr F is non-empty means
  :: ZF_REFLE:def 4
    not ∅ ∈ rng F;
end;
```

```
   proof
    hereby assume F is non-empty; then
```

The 'hereby' keyword behaves like the combination of 'thus' and 'now' (this construction is not in the official Mizar grammar from appendix B on page 16).

```
A1:    not ∅ ∈ rng F by ZF_REFLE:def 4;
     let i be set;
     assume i ∈ dom F;
     hence F.i is non empty by A1,FUNCT_1:11;
    end;
    assume
A2:    for n being set st n ∈ dom F holds F.n is non empty;
     assume ∅ ∈ rng F;
      then ex i being set st i ∈ dom F & F.i = ∅ by FUNCT_1:11;
     hence contradiction by A2;
   end;
end;

definition let A be non empty set; let x be Element of PFuncs(A*,A);
 redefine
  func <*x*> -> PFuncFinSequence of A;
 coherence
```

The correctness conditions for the redefinition of a 'func', are 'coherence' in case the type changes, and 'compatibility' in case the 'definition' changes. In this case only the first applies and the condition of course is:

```
<*x*> is PFuncFinSequence
```

```
  proof
    <*x*> is FinSequence of PFuncs(A*,A);
   hence thesis by Def3;
  end;
end;
```

```
definition let A be non empty set;
cluster homogeneous quasi_total non-empty PFuncFinSequence of A;
 existence
  proof
   consider a being Element of A;
   reconsider f = {<∅>A} -->a as PartFunc of A*,A by Th2;
   reconsider f as Element of PFuncs(A*,A) by PARTFUN1:119;
   take <*f*>;
   thus <*f*> is homogeneous
    proof
     let n; let h be PartFunc of A*,A; assume
     A1: n ∈ dom <*f*> & h =<*f*>.n;
     then n ∈ {1} by FINSEQ_1:4,FINSEQ_1:def 8; then
      h = <*f*>.1 by A1,TARSKI:def 1;
     then h = f & f is homogeneous PartFunc of A*,A by Th2,FINSEQ_1:def 8;
     hence thesis;
    end;
   thus <*f*> is quasi_total
    proof
     let n; let h be PartFunc of A*,A; assume
     A2: n ∈ dom <*f*> & h =<*f*>.n;
     then n ∈ {1} by FINSEQ_1:4,FINSEQ_1:def 8; then
      h = <*f*>.1 by A2,TARSKI:def 1;
     then h = f & f is quasi_total PartFunc of A*,A by Th2,FINSEQ_1:def 8;
     hence thesis;
    end;
   thus <*f*> is non-empty
    proof
     let n be set; assume
     A3: n ∈ dom <*f*>;
     then reconsider n as Nat;
     n ∈ {1} by FINSEQ_1:4,A3,FINSEQ_1:def 8; then
     n = 1 by TARSKI:def 1; then
   <*f*>.n=f by FINSEQ_1:def 8;
      hence thesis by Th2;
    end;
  end;
end;

reserve U for UAStr;

definition let IT be UAStr;
attr IT is partial means                                :Def7:
 the charact of IT is homogeneous;
attr IT is quasi_total means                            :Def8:
 the charact of IT is quasi_total;
attr IT is non-empty means                              :Def9:
 the charact of IT <> <∅> & the charact of IT is non-empty;
end;

reserve A for non empty set,
        h for PartFunc of A*,A  ,
        x,y for FinSequence of A,
        a for Element of A;

theorem Th4:
for x be Element of PFuncs(A*,A) st x = {<∅>A} --> a holds
  <*x*> is homogeneous quasi_total non-empty
 proof  let x be Element of PFuncs(A*,A) such that
  A1: x = {<∅>A} --> a;
```

The '`let` … `such that`' construction corresponds to a '`for` … `st`' in the statement, so it is equivalent to a '`let`' followed by an '`assume`'.

```
  reconsider f=x as PartFunc of A*,A by PARTFUN1:121;
  A2: for n,h st n ∈ dom <*x*> & h = <*x*>.n holds h is homogeneous
   proof let n,h; assume
    A3: n ∈ dom <*x*> & h =<*x*>.n;
    then n ∈ {1} by FINSEQ_1:4,FINSEQ_1:def 8; then
```

```
     h = <*x*>.1 by A3,TARSKI:def 1;
    then h = x & f is homogeneous PartFunc of A*,A by Th2,A1,FINSEQ_1:def 8;
    hence thesis;
   end;
  A4: for n,h st n ∈ dom <*x*> & h = <*x*>.n holds h is quasi_total
   proof let n,h; assume
    A5: n ∈ dom <*x*> & h =<*x*>.n;
    then n ∈ {1} by FINSEQ_1:4,FINSEQ_1:def 8; then
     h = <*x*>.1 by A5,TARSKI:def 1;
    then h = x & f is quasi_total PartFunc of A*,A by Th2,A1,FINSEQ_1:def 8;
    hence thesis;
   end;
  for n being set st n ∈ dom <*x*> holds <*x*>.n is non empty
   proof let n be set; assume
     n ∈ dom <*x*>;
    then n ∈ {1} by FINSEQ_1:4,FINSEQ_1:def 8; then
     <*x*>.n = <*x*>.1 by TARSKI:def 1;
    then <*x*>.n = x & f is non empty PartFunc of A*,A by Th2,A1
,FINSEQ_1:def 8;
    hence thesis;
   end;
  hence thesis by A2,A4,Def6,Def5,Def4;
 end;

definition
 cluster quasi_total partial non-empty strict non empty UAStr;
 existence
  proof
   consider A be non empty set;
   consider a be Element of A;
   set f = {<∅>A} --> a;
   reconsider w = f as Element of PFuncs(A*,A) by Th3;
   set U = UAStr ⟨⟨ A, <*w*> ⟩⟩;
   take U;
   A1: the charact of(U) is quasi_total &
      the charact of(U) is homogeneous & the charact of(U) is non-empty
    by Th4;
   the charact of(U) <> <∅>
    proof assume A2: the charact of(U) = <∅>;
     A3: len(the charact of(U)) = 1 by FINSEQ_1:56;
     len (<∅>) = 0 by FINSEQ_1:25;
     hence contradiction by A3,A2;
    end;
   hence thesis by A1,Def9,Def8,Def7;
  end;
end;

definition let U be partial UAStr;
 cluster the charact of U -> homogeneous;
 coherence by Def7;
end;

definition let U be quasi_total UAStr;
 cluster the charact of U -> quasi_total;
 coherence by Def8;
end;

definition let U be non-empty UAStr;
 cluster the charact of U -> non-empty non empty;
 coherence by Def9;
end;

definition
mode Universal_Algebra is quasi_total partial non-empty non empty UAStr;
end;

reserve U for partial non-empty non empty UAStr;

definition
  let A;
```

```
  let f be homogeneous non empty PartFunc of A*,A;
func arity(f) -> Nat means
 x ∈ dom f implies it = len x;
 existence
```

The correctness conditions of the definition of a func consists of an 'existence' and a 'uniqueness' part. In this case the 'existence' condition is:

```
ex n st
  for x st x ∈ dom f holds n = len x
```

```
  proof
   ex n st for x st x ∈ dom f holds n = len x
    proof
A1:   dom f <> ∅ by Th1;
     consider x being Element of dom f;
     dom f c= A* by RELSET_1:12; then
     x ∈ A* by A1,TARSKI:def 3; then
     reconsider x as FinSequence of A by FINSEQ_1:65;
     take n = len x;
     let y; assume y ∈ dom f;
     hence n = len y by Def1;
    end;
   hence thesis;
  end;
 uniqueness
```

The 'uniqueness' condition is:

```
for n,m st
  (for x st x ∈ dom f holds n = len x) &
  (for x st x ∈ dom f holds m = len x) holds
    n = m
```

```
  proof
     let n,m such that A2: (for x st x ∈ dom f holds n = len x) &
                     for x st x ∈ dom f holds m = len x;
A3:   dom f <> ∅ by Th1;
     consider x being Element of dom f;
     dom f c= A* by RELSET_1:12; then
     x ∈ A* by A3,TARSKI:def 3; then
     reconsider x as FinSequence of A by FINSEQ_1:65;
     n = len x & m = len x by A3,A2;
     hence thesis;
  end;
end;

theorem Th5:
for U holds for n st n∈dom the charact of(U) holds
  (the charact of(U)).n is
   PartFunc of (the carrier of U)*,the carrier of U
 proof let U,n;
  set pu = PFuncs((the carrier of U)*, the carrier of U),
      o = the charact of(U); assume
   n∈dom o; then
  o.n ∈ rng o & rng o c= pu by FUNCT_1:12,FINSEQ_1:def 4;
hence thesis by PARTFUN1:121;
 end;

definition let U;
func signature(U) ->FinSequence of NAT means
 len it = len the charact of(U) &
 for n st n ∈ dom it holds
   for h be homogeneous non empty
   PartFunc of (the carrier of U )*,the carrier of U
     st h = (the charact of(U)).n
    holds it.n = arity(h);
 existence
```

```
403   proof
404    defpred P[Nat,set] means
405    for h be homogeneous non empty
406       PartFunc of (the carrier of U )*,the carrier of U
407        st h = (the charact of(U)).$1
408         holds $2 = arity(h);
```

A ‘defpred’ defines a ‘local’ predicate: it’s like ‘set’, but for predicates instead of expressions. It’s expanded everywhere (this one is used in lines 415 and 419). The arguments are in the ‘body’ referred to as ‘$1’, ‘$2’, … Because the highest index allowed is ‘$8’, a ‘deffunc’ or ‘defpred’ takes at most eight arguments.

```
409    A1: now let m; assume
410         m∈ Seg len the charact of(U); then
411         m∈ dom the charact of(U) by FINSEQ_1:def 3; then
412         reconsider H=(the charact of(U)).m as homogeneous non empty
413          PartFunc of (the carrier of U )*,the carrier of U by Th5,Def4,Def6;
414         take n=arity(H);
415         thus P[m,n];
416        end;
417    consider p be FinSequence of NAT such that
418    A2: dom p = Seg(len the charact of(U)) and
419    A3: for m st m ∈ Seg(len the charact of(U)) holds P[m,p.m] from SeqDEx(A1);
```

The ‘SeqDEx’ scheme from article ‘MATRIX_2’ is defined as:

```
scheme SeqDEx{D()->non empty set,A()->Nat,P[set,set]}:
  ex p being FinSequence of D() st dom p = Seg A() &
    for k st k ∈ Seg A() holds P[k,p.k]
provided
  for k st k ∈ Seg A() ex x being Element of D() st P[k,x];
```

(Note that references to schemes are by name instead of by number: the Mizar library contains 546 schemes, less than the number of articles.) It is used here to derive:

```
ex p be FinSequence of NAT st
  dom p = Seg(len the charact of(U)) &
  for m st m ∈ Seg(len the charact of(U)) holds P[m,p.m]
```

(which is needed for the ‘consider’ statement), from:

```
A1: for m st m∈ Seg len the charact of(U) holds ex n st P[m,n]
```

So in this case the instantiation of the scheme is:

```
     D()  →  NAT
     A()  →  len the charact of U
  P[k,x]  →  P[k,x]
```

This substitution is not given explicitely, but is found by matching the ‘argument’ ‘A1’ with the ‘condition’ after the ‘provided’ in the scheme, and the statement to be proved with the ‘conclusion’ of the scheme.

```
420    take p;
421 Seg len the charact of(U) = dom the charact of(U) by FINSEQ_1: def 3;
422    hence A4: len p = len the charact of(U) by A2,FINSEQ_3:31;
423    let n; assume
424    n ∈ dom p; then
425    A5: n ∈ Seg(len the charact of(U)) by FINSEQ_1:def 3,A4;
426    let h be homogeneous non empty
427            PartFunc of (the carrier of U )*,the carrier of U; assume
428    h = (the charact of U).n;
```

```
   hence p.n = arity(h) by A5,A3;
  end;
 uniqueness
  proof
   let x,y be FinSequence of NAT; assume that
   A6: len x = len the charact of(U) and
   A7: for n st n ∈ dom x holds for h be homogeneous non empty
        PartFunc of (the carrier of U )*,the carrier of U
         st h = (the charact of(U)).n
         holds x.n = arity(h) and
   A8: len y = len the charact of(U) and
   A9: for n st n ∈ dom y holds for h be homogeneous non empty
         PartFunc of (the carrier of U )*,the carrier of U
          st h = (the charact of(U)).n
          holds y.n = arity(h);
    now let m; assume
     1≤m & m≤len x; then
     m ∈ Seg len x by FINSEQ_1:3; then
  m ∈ dom x by FINSEQ_1:def 3;
     then A10: m ∈ dom the charact of(U) & m∈dom x & m∈dom y
                                         by A6,A8,FINSEQ_3:31;
     then reconsider h=(the charact of(U)).m
       as homogeneous non empty
        PartFunc of (the carrier of U )*,the carrier of U by Th5,Def4,Def6;
     x.m=arity(h) & y.m=arity(h) by A7,A9,A10;
     hence x.m=y.m;
    end;
   hence thesis by A6,A8,FINSEQ_1:18;
  end;
end;
```