STUDIES IN LOGIC, GRAMMAR AND RHETORIC 18 (31) 2009

How to Define Terms in Mizar Effectively

Artur Korniłowicz

Institute of Computer Science
University of Białystok, Poland
arturk@math.uwb.edu.pl

Abstract. This paper explains how proofs written in Mizar can evolve if some dedicated mechanisms for defining terms are used properly, and how to write articles to fully exploit the potential of these mechanisms. In particular, demonstrated examples show how automatic expansion of terms and terms identification allow to write compact, yet readable proofs.

1 Introduction

It is commonplace that new authors writing their first Mizar [1] articles learn how to use the language and the verification system by looking at basic articles stored in the Mizar Mathematical Library. This is in principle the correct way, but the problem is that some of these articles were written many years ago, when many features currently available in Mizar had not been implemented yet. The Library Committee keeps revising the articles quite successively, rewriting proofs to use new features, but there is still much work to be done in this area. In fact it is probably a never ending process, since stronger and stronger mechanisms are still being added to the verifier to increase reasoning automation and make proofs much shorter, sometimes almost trivial.

We will take some articles written at the beginning of the previous decade as our working examples. We will focus particularly on automatic expansions of terms defined with the `equals` (this feature was implemented in August 2005 in Mizar version 7.6.01) and the identification of terms using `identify` (implemented in August 2006 in version 7.8.01).

2 Definitions

Mizar is an open language, i.e. users can introduce new symbols and define new notions. Symbols are qualified with the kinds of notions that can be defined, i.e. predicates, attributes, modes, functors, structures, selectors and left and right functorial brackets.

In this section we will focus on modes and functors only. In particular, we will give some hints on how their result types should be defined effectively.

Let us look at the following example, taken from [10].

```
definition
  let A, B be set;
  mode FUNCTION_DOMAIN of A,B -> functional non empty set means
:: FRAENKEL:def 2
  for x being Element of it holds x is Function of A,B;
end;
```

It defines a `functional non empty set`, named `FUNCTION_DOMAIN`, consisting of functions from any set `A` into a set `B`, where `functional` means that its every element is a function. Although at a first glance everything may look fine, we will analyze the result type of this definition in more detail.

The author's intention was to ensure that `FUNCTION_DOMAIN` is always non-empty. So the adjective `non empty` has been stated. This is correct. But the property `functional` can be proven based on the definiens, and therefore should not be a part of the result type. Of course, we would like to know that every `FUNCTION_DOMAIN` is `functional`. The best way to achieve that is to make a conditional registration:

```
registration
  let A, B be set;
  cluster -> functional FUNCTION_DOMAIN of A,B;
end;
```

Then the definition may be changed to:

```
definition
  let A, B be set;
  mode FUNCTION_DOMAIN of A,B -> non empty set means
  for x being Element of it holds x is Function of A,B;
end;
```

Why is this version accompanied with the registration better than the original definition? The point is seen when one introduces a functor with the result type `FUNCTION_DOMAIN`, like, for example, in [7]

```
definition
  let UA be Universal_Algebra;
  func UAAut UA ->
       FUNCTION_DOMAIN of the carrier of UA, the carrier of UA
  means
:: AUTALG_1:def 1
  for h being Function of UA, UA holds
      h in it iff h is_isomorphism UA, UA;
end;
```

To properly define functors, the Mizar verifier requires proving two correctness conditions: `existence` saying that there exists an object of the type `FUNCTION_DOMAIN` satisfying the definiens, and `uniqueness` saying that there exists exactly one such object. With the first condition in mind, it is obviously easier to construct an object with a less complex mother type, the “weaker” `FUNCTION_DOMAIN` is more convenient, and `non empty set` is obviously “weaker” than `functional non empty set`. One disadvantage of this approach is that the extra conditional registration must always be imported to fully exploit the definition. But once we have the registration imported the checker knows that not only `UAAut`, but all other functors with the result type `FUNCTION_DOMAIN` are `functorial` without any proof. And this is the real gain in the end.

A slightly different situation is when one introduces a new functor. The result type of a functor must contain the information needed to formulate its definiens in a natural and concise way and to allow proving its `uniqueness`. For example, in

```
definition
  let f1,f2 be complex-valued Function;
  func f1 + f2 -> Function means
:: VALUED_1:def 1
  dom it = dom f1 /\ dom f2 &
  for c being set st c in dom it holds it.c = f1.c + f2.c;
end;
```

`f1+f2` must be of the type at least `Function` since the application operator (`.`), which is defined for functions, is applied to it.

A typical example showing that some adjectives are required in the mother type of a functor is a definition of a structural object, where needed selectors are described. For example, in the definition of the product of two relational structures (see [6])

```
definition
  let X, Y be RelStr;
  func [:X,Y:] -> strict RelStr means
:: YELLOW_3:def 2
  the carrier of it =
      [:the carrier of X, the carrier of Y:] &
  the InternalRel of it =
      ["the InternalRel of X, the InternalRel of Y"];
end;
```

the adjective `strict`, saying that there are no other fields in the defined structure, is necessary to prove the `uniqueness` condition. `RelStr` is a relational structure which can be a predecessor of, for example, a relation structure extended by a topology. So, it is clear that not all relational structures with the carrier and the internal relation described above are equal. But if the structures are both `strict`, they are equal.

3 Theorems and Registrations

In this section we demonstrate what theorems and registrations should be stated in an article to achieve short proofs and use the full Mizar power.

As an example let us take a proof of the associativity of the supremum in the lattice of natural numbers with the GCD and LCM operations taken from [4].

A very naive proof which does not use too many Mizar features would look like:

```
registration ::R1
  cluster Nat_Lattice -> join-associative;
  coherence
  proof
    let p, q, r be Element of Nat_Lattice;
    set L = the L_join of Nat_Lattice;
    set o = lcmlat;
a1: L = o by Def5;
    reconsider p1 = p, q1 = q, r1 = r as Element of NAT by Def5;
    thus p"\/"q"\/"r = L.(p"\/"q,r) by LATTICES:def 1
    .= L.(L.(p,q),r) by LATTICES:def 1
    .= o.(p1 lcm q1,r) by a1,Def4
    .= p1 lcm q1 lcm r1 by Def4
    .= p1 lcm (q1 lcm r1) by NEWTON:56
    .= o.(p,q1 lcm r1) by Def4
    .= L.(p,L.(q,r)) by a1,Def4
    .= L.(p,q"\/"r) by LATTICES:def 1
    .= p"\/"(q"\/"r) by LATTICES:def 1;
  end;
end;
```

where `Def4` is a definition of the operation playing role of the supremum and `Def5` is a definition of the lattice, both taken from [4].
The first step to make the proof shorter is to introduce a theorem showing the correspondence between the operation in the lattice and the operation on numbers, like:

```
theorem T0:
  for x, y being Element of Nat_Lattice
  for m, n being Nat st x = m & y = n holds
  x "\/" y = m lcm n
  proof
    let p, q be Element of Nat_Lattice;
    let p1, q1 be Nat such that
a1: p = p1 & q = q1;
    thus p"\/"q = (the L_join of Nat_Lattice).(p,q)
                  by LATTICES:def 1
    .= lcmlat.(p,q) by Def5
    .= p1 lcm q1 by a1,Def4;
  end;
```

which gives the more concise proof of the registration

```
registration ::R2
  cluster Nat_Lattice -> join-associative;
  coherence
  proof
    let p, q, r be Element of Nat_Lattice;
    reconsider p1 = p, q1 = q, r1 = r as Element of NAT by Def5;
a1: q"\/"r = q1 lcm r1 by T0;
    p"\/"q = p1 lcm q1 by T0;
    hence p"\/"q"\/"r = p1 lcm q1 lcm r1 by T0
    .= p1 lcm (q1 lcm r1) by NEWTON:56
    .= p"\/"(q"\/"r) by a1,T0;
  end;
end;
```

Observe that because the operator `lcm` can only be applied to numbers, extra variables `p1`, `q1` and `r1` are needed inside the proof to switch from the lattice context to numbers, and vice versa. In such a situation the conditional registration

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

helps to make it even simpler. It allows to treat elements of the lattice as natural numbers, and then the theorem `T0` can be reformulated as:

```
theorem T1:
  for x, y being Element of Nat_Lattice holds x "\/" y = x lcm y
  proof
    let p, q be Element of Nat_Lattice;
    thus p"\/"q = (the L_join of Nat_Lattice).(p,q)
                  by LATTICES:def 1
    .= lcmlat.(p,q) by Def5
    .= p lcm q by Def4;
  end;
```

and the registration `R2` as:

```
registration ::R3
  cluster Nat_Lattice -> join-associative;
  coherence
  proof
    let p, q, r be Element of Nat_Lattice;
a1: q"\/"r = q lcm r by T1;
    p"\/"q = p lcm q by T1;
    hence p"\/"q"\/"r = p lcm q lcm r by T1
    .= p lcm (q lcm r) by NEWTON:56
    .= p"\/"(q"\/"r) by a1,T1;
  end;
end;
```

In this case less variables are needed, which clearly makes the proof easier.

Another feature which increases the deduction power of the MIZAR checker is automatic expansion of terms defined using the `equals` keyword. In the above example, several times we explicitly referred to the definition:

```
definition
  let G be non empty \/-SemiLattStr, p, q be Element of G;
  func p "\/" q -> Element of G equals
:: LATTICES:def 1
  (the L_join of G).(p,q);
end;
```

However, this is not really necessary. It is enough to extend the article's environment by the directive `definitions LATTICES`, where the notion was defined, see [12]. With this directive in effect, the checker will always automatically equate all occurrences of `"\/"` with `the L_join`. That process makes the theorem `T1` almost obvious:

```
theorem
  for x, y being Element of Nat_Lattice holds
  x "\/" y = x lcm y by Def4;
```

please note that even if the proof is trivial, the theorem should be stated and referred to when needed. But one of the newest enhancements of the MIZAR checker makes even such references unnecessary by identifying selected terms internally. This feature is discussed in the next section.

4 Terms Identification

In mathematical practice, a given object or an operation is often treated in many different ways depending on contexts in which they occur. A natural number can be considered as a number, or as a finite ordinal. The least common multiply can be considered as an operation on numbers, or as the supremum of elements of some lattice, as we do in our example. In such cases it is often worthwhile to have 'translation' theorems, like the last one in the previous section. But it would be really comfortable for the users, if they did not have to refer to such theorems, but rather had them 'built-in' some way. The developers of MIZAR decided to implement such a feature, naming it *terms identification*, and introducing a new keyword, `identify`, in the MIZAR language. Its syntax is the following:

```
Identify-Registration =
"identify" Functor-Pattern "with" Functor-Pattern
[ "when" Variable-Identifier "=" Variable-Identifier
   { "," Variable-Identifier "=" Variable-Identifier } ] ";"
Correctness-Conditions .
```

where `Identify-Registration` is a part of the rule

```
Registration-Block = "registration"
{ Loci-Declaration | Cluster-Registration | Identify-Registration }
"end" .
```

and `Correctness-Conditions` is `compatibility` described later.

The aim of the identification is matching the term at the left side of the `with` keyword with the term stated at the right side, whenever they occur together. The current implementation (version `7.11.01`) allows matching in one direction only, i.e. when the verifier processes a sentence containing the left side term, it generates its local copy with the left side term symbol substituted by the right side one and makes both terms equal to each other. Such a equality allows to justify facts about the left side terms via lemmas written about the right side ones, but not vice versa. In this sense identification is not symmetric, which is showed with the following example. First, we introduce the registration:

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

The lemma

```
L1: for x, y, z being Element of Nat_Lattice holds
    x lcm y lcm z = x lcm (y lcm z);
```

can be used to justify the sentence

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

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

Let us now see at the proof of the associativity of the supremum that uses this mechanism:

```
registration
  cluster Nat_Lattice -> join-associative;
  coherence
  proof
    let p, q, r be Element of Nat_Lattice;
    thus thesis by NEWTON:56;
  end;
end;
```

Comparing it with `R3`, the power of terms identification becomes evident.

4.1 Some Technical Aspects

Correctness Conditions Terms identification can be used to identify two terms built with different functor symbols, but also with the same symbol. In the case when different variables are used at both sides of `with`, a `when` clause must be used to establish the correspondence between appropriate arguments. Depending on whether the `when` clause occurs or not, the system generates two different conditions:

```
registration
  let p, q be Element of Nat_Lattice;
  let m, n be Nat;
  identify p "\/" q with m lcm n when p = m, q = n;
  compatibility
  proof
    thus p = m & q = n implies p "\/" q = m lcm n;
  end;
end;

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

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

4.2 Typical Errors Reported

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

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

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

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

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

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

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

4.3 Examples of Use

Here we list some natural and useful identifications introduced in the Mizar Mathematical Library.

```
registration
  let a, b be Element of G_Real, x,y be real number;
  identify a+b with x+y when a = x, b = y;
end;

registration
  let a be Element of G_Real, x be real number;
  identify -a with -x when a = x;
end;
```

registered in [8], where G_Real is the additive group of real numbers.

```
registration
  let a, b be Element of Real_Lattice;
  identify a "\/" b with max(a,b);
  identify a "/\" b with min(a,b);
end;
```

registered in [5], where Real_Lattice is the lattice of real numbers with the max and min operations.

```
registration
  let a, b be Element of INT.Group;
  identify a*b with a+b;
end;
```

introduced in [9], where `INT.Group` is the additive group of integers.

`identify` can be used not only when structural objects are constructed, but also in, so called, classical cases:

```
registration
  let X, D be non empty set,
      p be Function of X,D, i be Element of X;
  identify p/.i with p.i;
end;
```

registered in [2], where `p/.i` is a restricted application, defined in [3].

```
registration
  let p be XFinSequence;
  identify len p with dom p;
end;
```

introduced in [11], where `len` stands for the length and `dom` for the domain of a finite sequence.

```
registration
  let x, y be real number, a, b be complex number;
  identify x+y with a+b when x = a, y = b;
  identify x*y with a*b when x = a, y = b;
end;
```

defined in XXREAL_3, where `x+y` and `x*y` are operations on extended reals and `a+b` and `a*b` are defined for complex numbers.

5 Conclusions

When a new feature is implemented in a system coupled with a database of source files, like Mizar and the Mizar Mathematical Library, it is desirable to 're-write' the database to exploit new possibilities. But in general it is not possible to find all contexts where these new features could be used, e.g. it is an undecidable problem whether two terms are equal or not, and then no automatic tools can be invented to find all cases where terms identification could be registered. Therefore, on behalf of the Mizar Library Committee, with this paper we would like to issue an appeal to Mizar users to get more involved in the continuous process of Mizar Mathematical Library revisions motivated either by finding better ways of formalization of some facts, or by implementation of stronger mechanisms in the checker. The simplest thing that all users could do is to report what useful revisions can, or should, be processed.

We showed in this paper that terms identifications definitely make proofs more compact: the original proof of associativity of the supremum was 14 lines long, while the new one using automatic expansion of terms and terms identification had only 2 lines. So it would be valuable to gather from all Mizar users more information on what terms identifications of notions already defined in the Mizar Mathematical Library could be registered.

References

1. Mizar homepage: `http://mizar.org`.
2. Czesław Byliński. Functions from a Set to a Set. *Formalized Mathematics*, 1(**1**):153–164, 1990. MML Id: `FUNCT_2`.
3. Czesław Byliński. Partial Functions. *Formalized Mathematics*, 1(**2**):357–367, 1990. MML Id: `PARTFUN1`.
4. Marek Chmur. The Lattice of Natural Numbers and The Sublattice of it. The Set of Prime Numbers. *Formalized Mathematics*, 2(**4**):453–459, 1991. MML Id: `NAT_LAT`.
5. Marek Chmur. The Lattice of Real Numbers. The Lattice of Real Functions. *Formalized Mathematics*, 1(**4**):681–684, 1990. MML Id: `REAL_LAT`.
6. Artur Korniłowicz. Cartesian Products of Relations and Relational Structures. *Formalized Mathematics*, 6(**1**):145–152, 1997. MML Id: `YELLOW_3`.
7. Artur Korniłowicz. On the Group of Automorphisms of Universal Algebra & Many Sorted Algebra. *Formalized Mathematics*, 5(**2**):221–226, 1996. MML Id: `AUTALG_1`.
8. Eugeniusz Kusak, Wojciech Leończuk, and Michał Muzalewski. Abelian Groups, Fields and Vector Spaces. *Formalized Mathematics*, 1(**2**):335–342, 1990. MML Id: `VECTSP_1`.
9. Dariusz Surowik. Cyclic Groups and Some of Their Properties – Part I. *Formalized Mathematics*, 2(**5**):623–627, 1991. MML Id: `GR_CY_1`.
10. Andrzej Trybulec. Function Domains and Frænkel Operator. *Formalized Mathematics*, 1(**3**):495–500, 1990. MML Id: `FRAENKEL`.
11. Tetsuya Tsunetou, Grzegorz Bancerek, and Yatsuka Nakamura. Zero-Based Finite Sequences. *Formalized Mathematics*, 9(**4**):825–829, 2001. MML Id: `AFINSQ_1`.
12. Stanisław Żukowski. Introduction to Lattice Theory. *Formalized Mathematics*, 1(**1**):215–222, 1990. MML Id: `LATTICES`.