# Enhanced Processing of Adjectives in Mizar

Adam Naumowicz

Institute of Computer Science
University of Białystok, Poland
adamn@mizar.org

**Abstract.** As adjectival notions are ubiquitous in informal mathematics, their important role must also be reflected in formal attempts to reconstruct the existing body of mathematical texts. In this paper we describe an enhancement of the Mizar proof checker which enables a more complete automation of notions encoded as adjectives. The proposed improvement concerns the Equalizer – Mizar's module responsible for handling equality, where adjective registrations can be re-used by matching them with classes of equal terms in order to add extra information to inference steps.

## 1 Introduction

It is common knowledge that successful formalization of mathematical texts requires the underlying formal language be near to the intuition of a mathematician. The fine details of mathematics are much better accounted for in a mixture of mathematical and natural language than in a purely set-theoretical setting. Therefore the languages devised particularly for formalization support the linguistic categories like adjectives that are amply used in informal texts, although they may seem superfluous from the formal point of view. The support for adjectives in the Mizar language dates back to 1983/84, when a version called Mizar HPF (with hidden parameters and functions) was implemented to facilitate a much richer syntax [3]. The idea was also present in de Bruijn's famous Mathematical Vernacular, and is also considered a key feature of its modern derivatives like Wtt [2].

In most (natural) languages that support adjectives, they form an open class of words, i.e. it is relatively common for new adjectives to be formed via derivation. In the formal context, this usually means applying 'technical' suffixes like '-like' or prefixes like 'being_', 'having_', or 'with_' to predicates [3]. When attributes were introduced in Mizar, such changes were done semi-automatically to numerous predicates previously defined in the Mizar library[1].

Since that time, adjectives have been playing a more and more important role helping to minimize information losses in Mizar formalizations. There is still, however, an open area for research on how to improve the processing of adjectival notions in Mizar to imitate even better the use of adjectives in real mathematics. This paper describes an attempt going in this direction.

[1] See also Andrzej Trybulec's posting to the Mizar-Forum mailing list on this topic: http://mizar.uwb.edu.pl/forum/archive/0006/msg00003.html .

ISBN 978-83-7431-229-5 ISSN 0860–150X

## 2 Registrations of adjective clusters

The paper [6] presents a detailed discussion on how adjectives are handled in Mizar. Let us just recall here that a “cluster of adjectives” is a collection of attributes (constructors of adjectives) with boolean values associated with them (negated or not) and their arguments. The tree-like hierarchical structure of Mizar types is built by the widening relation which uses such collections of adjectives to extend existing types [1, 7]. Grouping adjectives in clusters enables automation of some type inference rules. Such rules are encoded in the form of so called registrations. Previously proved registrations can subsequently be used to secure the non-emptiness of Mizar types (existential registrations), to allow formulating and automating relationships between adjectives (conditional registrations) and to store adjectives that are always true for instantiations of terms with certain arguments (functorial registrations), cf. [6].

Before the proposed enhancement, the role of adjective processing was mostly syntactic, i.e. the Analyzer automatically “rounded-up” the information from all available registrations to disambiguate used constructors and check their applicability. See [5] for a more detailed overview of the internal functioning of the Mizar `verifier` and the tasks distributed among the Analyzer and the Checker (in particular the Equalizer). The semantic role was restricted to processing only the type information for the terms explicitely stated in an inference. Also, attributive statements as premises or conclusions were not “rounded-up”. Neither did the automation take into account the potential of applying registrations to every element of a class of equal terms generated in the Equalizer as a consequence of the equality calculus.

## 3 Examples

In this section we present some contexts where the automatic type reasoning did not work, while it seemed rather “natural” to expect such statement be accepted by the Checker. The most typical situation concerns attributive statements (`x is <some-attribute>`) or qualifications (`x is <some-type>`) among the premises. Even with the following registration imported from the article `RAT_1`[2]

```
registration
  cluster integer -> rational number;
end;
```

the last line in the listing below was not automatically accepted (the statement was marked with the `::> *4` flag by the Checker):

[2] Unique identifiers, like `RAT_1` in this case, are assigned to all Mizar articles stored in the MML. The respective files can be found in the `mml` sub-directory of the Mizar distribution.

```
now
  let a be integer number;
  let b be number;
  a is rational;
  b is integer implies b is rational;
::>                               *4
end;
```

Although the Checker accepted that `a is rational`, it was not the case for `b`, because the adjective `integer` was only mentioned as a premise and not in the variable declaration, so it was not “rounded-up”.

The next example shows an inference where the type of neither constant can be “rounded-up” on its own with a registration taken from `XXREAL_0`, but after processing the equality `a=b` the statement should “naturally” be obvious (because `a` and `b` should share all adjectives as elements of the same equality class):

```
registration
  cluster non negative non zero ->
                    positive (ext-real number);
end;

now
  let a be non negative (real number);
  let b be non zero number;
  a=b implies a is positive;
::>                       *4
end;
```

The enhancement presented in Section 4 makes the Checker accept such statements. Moreover, please note that this more complete “rounding-up” of attributive statements gives also a nice side efect: in the Checker we get something like a contraposition of conditional registrations, because the conclusion is negated when the Checker tries to deduce contradiction, cf. [3].

In this sense registrations are used like theorems, when it is enough to state an implication and then be able to use it in a contraposition form. Of course it does not mean that registering contraposition forms of conditional registrations is not needed anymore – sometimes we want the Analyzer to use them as well (e.g. to match certain notations).

To show an iterative effect of the “rounding-up” done for more complicated terms, we may look at two functorial registrations extracted from the article `ABIAN`:

```
registration
  let i be even Integer, j be Integer;
  cluster i*j -> even;
end;

registration
  let i be even Integer, j be odd Integer;
  cluster i+j -> odd;
end;
```

With these registrations imported, all the following statements are now accepted by the enhanced Checker while the old realization would mark them all as unaccepted:

```
now
   let i,e,o be integer number;
   e is even implies i*e is even;
   e is even & o is odd implies e+o is odd;
   e is even & o is odd implies (i*e)+o is odd;

   let z be complex number;
   z is even Integer implies z*i is even;
end;
```

Let us note that in the above examples all the attributes are absolute, i.e. their only argument is the subject. But in general, the subject may be defined with a type that has its own (explicit or implicit) arguments, and so the adjective has more implicit arguments. For example, with the following definition of continuity as defined in the article PRE_TOPC:

```
definition
  let S,T be TopStruct, f be Function of S,T;
  attr f is continuous means
  for P1 being Subset of T st
          P1 is closed holds f" P1 is closed;
end;
```

the function f may also map S to some space T1, and then may not be continuous with respect to S and T1, so the processing of arguments is crucial to a proper "rounding-up" extension in the Equalizer.

## 4 The extended "round-up" algorithm

The motivation for this work was presented in [8] showing the limitation of the adjective handling process. As mentioned in Section 2, previously clusters of adjectives in types were "rounded-up" in the Analyzer. The Equalizer just "adjusted" the clusters, trying to move adjectives between various types of the same equality constant if they were applicable. The extended algorithm presented below takes into account the complex structure of terms processed in the Equalizer, that may have numerous representatives, as well as multiple types, which in turn have their arguments of the same form, and so on.

As a class may have several types and several term instances that may match the same registration, the result of matching is a list of instantiations of classes for the loci used in a registration (usually it is just one instantiation, but sometimes there are more, see Section 5 for maximal values noted with current MML).

Technically, the implementation reuses some of the data structures already developed for the Unifier, where an algebra of substitutions is used to contradict a given universal formula, cf. [8]. The main difference is when joining instantiation lists – in the Unifier longer substitution is absorbed, while here the longer substitution remains.

In the calculus of (lists of) instantiations we will use two binary functions, `JOIN` and `MEET` with the following semantics:

- `JOIN(l1,l2)` produces a union of lists `l1` and `l2`, replacing shorter substitutions with longer ones – unlike in the Unifier, where a shorter list is always preferred as it is used for refutation ([8]);
- `MEET(l1,l2)` produces a collection of unions of two instantiations (one from `l1`, the other from `l2` provided they agree on the intersection of their domains; again a shorter substitution is replaced by a longer one if they both are inserted into this collection).

For convenience we also use two lattice-like constants: `TOP` which denotes a trivial substitution (no loci to be substituted, but all constants are matched) and `BOTTOM` which is an empty list of substitutions (no match found). Our `TOP` and `BOTTOM` have the usual lattice properties, e.g. are neutral with respect to the `MEET` and `JOIN` operations, respectively.

Below is an outline of pseudo-code functions that have been implemented to enable matching a class of terms with a given registration. All these functions return as their result a (possibly empty) list of substitutions of classes for loci in the registration. It should be clear which of the `match` functions is used in a certain context looking at the type of their arguments. For simplicity, we treat any class of terms `E` as a special kind of a term – one that satisfies the condition `E is CLASS`.

Let us consider a conditional registration `C`. To check if a given class `E` matches `C` we generate substitutions which match both the type and the antecedent of `C`:

```
match(E:term,C:condreg)
begin
  l:=match(E,C.type)
  l:=MEET(l,match(E,C.antecedent))
  return l
end
```

In the case of a functorial registration F, the matching function generates substitutions which match both the registered type and term of F. As above, if a substitution is found, it can be used to extend the cluster of the equality class E. Let us note that F.type is just a radix type, the adjectives from the type's cluster of adjectives do not have arguments other than that of the type, so the cluster does not have to be matched as such:

```
match(E:term,F:funcreg)
begin
  l:=match(E,F.type)
  l:=MEET(l,match(E,F.term))
  return l
end
```

Matching a class E with a type T is just matching one by one all the types of E with T:

```
match(E:term,T:type)
begin
  l:=BOTTOM
  if E is CLASS then
    for t in E.types do
      l:=JOIN(l,match(t,T))
  return l
end
```

When types T1 and T2 are to be matched, they must denote the same mode (T1.id=T2.id) as well as all their arguments must match:

```
match(T1:type,T2:type)
begin
  if T1.id=T2.id then
    begin
      l:=TOP
      while n do
        l:=MEET(l,match(T1.arg(n),T2.arg(n)))
    end
  else return BOT
  return l
end
```

Matching terms is the main part of the substitution process, since terms are arguments of terms, types and adjectives. Therefore, all matching must eventually come to this point. A class `E` can be matched with a term `T` being a locus in a registration if the type of `T` (`T.type`) and the cluster of adjectives of `T` (`T.cluster`) match the class `E`. Having a valid substitution, we merge it with `(T<-E)` (`E` is substituted for `T`).

If `E` is a class but `T` is not a locus, then we generate a union of possible matches of instances of `E` (taken from `E.terms`) with `T`.

Otherwise, if `E` and `T` have the same kind and number (so `E` is not a class and `T` is not a locus), then we simply match all their arguments:

```
match(E:term,T:term)
begin
  if E is CLASS then
    begin
      if T is LOCUS then
        begin
          l:=match(E,T.type))
          l:=MEET(l,match(E,T.cluster))
          l:=MEET(l,(T<-E))
          return l
        end
      else
        begin
          l:=BOTTOM
          for t in E.terms do l:=JOIN(l,match(t,T))
          return l
        end
    end
  else
    if E.id=T.id then
      begin
        l:=TOP
        while n do
          l:=MEET(l,match(E.arg(n),T.arg(n)))
        return l
      end
    else return BOTTOM
end
```

Matching a class `E` with a cluster of adjectives (for matching an antecedent of a conditional registration or a cluster accompanying the type of a locus) can be split for clarity into the following two steps:

```
match(E:term,L:cluster)
begin
  l:=TOP
  for a in L.adjectives do
    l:=MEET(l,match(E,a))
  return l
end
```

and finally matching single adjectives as below. An adjective `A` matches some adjective in the cluster of a class `E` (`E.cluster`) if they denote the same attribute, have the same boolean value, and their arguments match as well:

```
match(E:term,A:adjective)
begin
  l:=BOT
  if E is CLASS then
    for a in E.cluster do
    if a.id=A.id & a.bool=A.bool then
      begin
        l1:=TOP
        while n do
          l1:=MEET(l1,match(a.arg(n),A.arg(n)))
        l:=JOIN(l,l1)
      end
  return l
end
```

With the matching function as above, we can now present the actual “round-up” algorithm as follows:

0. Create a dependence list for all equivalence classes in a given inference. Let `dep(E)` denote a list of all classes in which `E` appears as a term argument.
1. Put all classes into a set `CLASSES`
2. Proceed as below until `CLASSES` remains empty:

```
while CLASSES <> {} do
  begin
    take E from CLASSES
    repeat
      extended=:false
      for C in CondRegs do
        l:=match(E,C)
        if l<>BOTTOM then
          begin
            extend E.cluster with l
              applied to C.consequent
            extended:=true
          end
      for F in FuncRegs do
        l:=match(E,F)
        if l<>BOTTOM then
          begin
            extend E.cluster with l
              applied to F.consequent
            extended=true
          end
     if extended then
       CLASSES=CLASSES+dep(E)
    until not extended
  end
```

As we see, if the class is matched with some registration, the resulting substitution is used to substitute all loci in the registration's consequent cluster with appropriate classes, this new cluster should be used to extend the cluster of a given class. Please note that "rounding-up" with a conditional registration may enable a functorial one, and the other way round, so the `repeat` loop must guarantee that the algorithm is not sensitive to the order of registrations.

## 5 Some statistics

Below we present the statistics collected by running main Mizar utility programs (`verifier`, `relprem`, `trivdemo`, and `relinfer`) equipped with the strengthened Checker as compared to the current official system to demonstrate the performance of the new algorithm. All tests have been performed using Mizar ver. 7.9.01 and MML ver. 4.103.1019 on a single-processor Intel Xeon 2.8 GHz machine running under the Linux operating system.

Here are the number of calls to the "round-up" procedure for respective utilities:

| | |
|---|---|
| `verifier` | 860379 |
| `relprem` | 3128080 |
| `trivdemo` | 182984 |
| `relinfer` | 700725 |

More detailed statistics concerning the equality classes occurring in all inferences were collected for the `verifier`. The biggest number of instantiations generated when "rounding-up" a functorial registration was 17 (in `POLYEQ_3`) and 9 in the case of conditional registrations (in `LIMFUNC2`). The average number of equality classes was 30.366 (max. 828 in `SCMISORT`). The average number of conditional registrations tried was 52.2439 (max. 193 in `JORDAN`) and 163.911 for functorial registrations (max. 671 in `AOFA_I00`). The average number of terms in equality classes was 1.60257. Here we may notice that the biggest number (182) was detected in `FILTER_1` resulting from numerous instances of equal terms built from automatically expanded commutative lattice operations.

The tools `relprem`, `trivdemo` and `relinfer` work with a different population of inferences, so the numbers were slightly different. However, the differences in the case of `relprem` were very small. A noticeable difference was the biggest number of equality classes when running `trivdemo` (728 in `SCMISORT` with a slightly smaller average of 27.7032, and the biggest number of classes for `relinfer`: 826 in `SCMISORT`) with the average of 39.3855 classes.

The extra code to be executed in order to "round-up" the adjectives in the Checker caused a noticeable slowdown in all the utilities. In the table below we present the calculated total running times for the selected utilities applied to the whole library, with the effective times put in brackets (without the extra time needed to load the utilities, accommodate each file, etc.).

| Program | Time (effective) | New time | Ratio |
|---|---|---|---|
| `verifier` | 2h 40m (8791s) | 4h 48 m (16486s) | 1.8753 |
| `relprem` | 16h 43m (59291s) | 24h 38m (87812s) | 1.4810 |
| `trivdemo` | 2h 44m (9053) | 3h 17m (10977s) | 1.2125 |
| `relinfer` | 7h 21m (22209s) | 9h 27m (29975s) | 1.3496 |

In the following table we compare the times of running the Checker pass only.

| Program | Checker | New Checker | Ratio |
|---|---|---|---|
| `verifier` | 5544s | 13131s | 2.3685 |
| `relprem` | 56029s | 84490s | 1.5079 |
| `trivdemo` | 5762s | 7304s | 1.2676 |
| `relinfer` | 22068s | 29839s | 1.35214 |

As far as single articles are concerned, below are tables showing the 10 articles with the longest processing time (in seconds) for each of the utilities. Here is the data collected for `verifier` and `relprem`:

| Secs. | `verifier` | Secs. | New `verifier` |
|---|---|---|---|
| 51 | BVFUNC_6 | 116 | GROUP_9 |
| 54 | GLIB_004 | 116 | AOFA_000 |
| 57 | JGRAPH_4 | 117 | SCMFSA10 |
| 65 | SCMISORT | 124 | JGRAPH_4 |
| 85 | JORDAN | 144 | SCMBSORT |
| 90 | BINARITH | 226 | JORDAN |
| 93 | BINARI_2 | 237 | SCPQSORT |
| 105 | AOFA_I00 | 240 | AOFA_I00 |
| 125 | SCPQSORT | 240 | SCMISORT |
| 168 | QUATERN2 | 409 | QUATERN2 |

| Secs. | `relprem` | Secs. | New `relprem` |
|---|---|---|---|
| 913 | JORDAN1G | 1163 | SCPQSORT |
| 973 | JORDAN1J | 1236 | GEOMTRAP |
| 1238 | GEOMTRAP | 1428 | JORDAN |
| 1276 | GLIB_005 | 1434 | GLIB_005 |
| 1428 | SCMFSA10 | 1537 | CONMETR |
| 1485 | JORDAN15 | 1644 | AFF_2 |
| 1529 | CONMETR | 1718 | JORDAN15 |
| 1656 | AFF_2 | 1861 | SCMFSA10 |
| 1694 | JORDAN19 | 1969 | JORDAN19 |
| 3972 | CONMETR1 | 3991 | CONMETR1 |

Below is the list of the 10 articles that need the longest time to get processed by `trivdemo` and `relinfer`:

| Secs. | `trivdemo` | Secs. | New `trivdemo` |
|---|---|---|---|
| 85 | JORDAN | 101 | SCPQSORT |
| 108 | AOFA_I00 | 114 | JORDAN |
| 113 | JORDAN15 | 117 | JORDAN15 |
| 118 | GEOMTRAP | 118 | GEOMTRAP |
| 124 | JORDAN19 | 128 | JORDAN19 |
| 133 | GATE_3 | 133 | GATE_3 |
| 160 | GATE_4 | 157 | GATE_4 |
| 161 | SCMFSA_2 | 166 | AOFA_I00 |
| 200 | ANALORT | 200 | ANALORT |
| 770 | XBOOLEAN | 763 | XBOOLEAN |

| Secs. | `relinfer` | Secs. | New `relinfer` |
|---|---|---|---|
| 376 | SCPQSORT | 425 | CONMETR |
| 385 | ANALMETR | 470 | AOFA_I00 |
| 442 | CONMETR | 501 | GEOMTRAP |
| 457 | JORDAN | 567 | SCPQSORT |
| 520 | GEOMTRAP | 663 | JORDAN |
| 652 | JORDAN19 | 694 | CONMETR1 |
| 674 | JORDAN15 | 729 | JGRAPH_7 |
| 694 | CONMETR1 | 754 | JORDAN19 |
| 703 | JGRAPH_7 | 770 | JORDAN15 |
| 919 | ANALORT | 926 | ANALORT |

One may notice that in most cases the same articles appear in the top 10 lists for each utility – it shows that the new extension works in a quite predictable way.

Note also that in some cases the running times of the new Checker are even smaller, e.g. `SCMFSA_2` ranking high for `trivdemo` (160 secs.) does not appear in the list of the new `trivdemo`. It is because the new Checker, in this case in `trivdemo`, is able to find much quicker two proofs that can be reduced to a straightforward justification and then it runs for just 40 secs.

The benefit of applying the new Checker was measured by the number of errors reported by the standard utilities after running it on all MML articles and then the number of bytes removed from the library when these errors had been corrected. The experiment was performed on the library "clean" with respect to `relprem` and `trivdemo` (also `inacc` and `chklab`). As there is currently no tool available to handle `relinfer` errors reliably and in a fully automatic way, the results for `relinfer` are calculated as the difference between the number of errors reported by the new and the old version.

Totally, errors were detected in 771 articles: 720 for `relprem`, 261 for `trivdemo` and 637 for `relinfer`. The `relprem` utility detected 8439 errors (there were 296 errors in JGRAPH_4) `trivdemo` detected 580 + 20 errors[3] (there were 36 errors in TEX_2) and `relinfer` 6555 (10834 – 4279) errors (also note that there were 392 errors in JGRAPH_4).

Only by automatic elimination of `relprem` and `trivdemo` errors, the size of MML was reduced from the original 72826045 bytes to 72587040 bytes. Although the difference (239005 bytes) is only 0.33% of the total size, one may see that with the average size of one MML article equal 71468.2 bytes this makes 3.3 articles. Moreover, with a very rough estimate of 20 bytes reduction per one `relinfer` error, that would give another 130KB, i.e. two more articles. The most significant changes were done to the articles JGRAPH_4 (4593 bytes removed) and TEX_2 (13483 bytes removed).

All in all, admitting the slowdown in processing time, we claim that the new Checker seems to work in a predictable way and the reduction of Mizar texts obtained thanks to it seems to make up for the loss.

## 6 Further work

The matching functions of the extended "round-up" algorithm presented in Section 4 have been written to demonstrate how the rounding-up should work rather than provide a final and most efficient solution. Eventually, profiling methods should be used to identify the weak points of the rudimentary implementation and a more efficient code with indexing and other optimizations should be developed to speed up the processing.

The statistics presented in Section 5 show the immediate benefits of running the strengthened Checker on the articles collected in MML at the time of writing this article. But it will also be very important to make a proper use of the new feature when developing new articles as well as during future MML revisions.

It would also be highly desirable to develop tools suitable for processing all MML articles in order to detect and eliminate cases of unnecessary `reconsider` statements and their proofs, because very often Mizar authors used to state an explicite type restriction only to help the Checker use a specific registration, while with the new "rounding-up" mechanism it is not needed anymore.

And finally, as always when the system has been strengthened, it would be very interesting to turn off the relevant code after some time, and check if authors make use of it in their new developments. Hopefully, extended adjective techniques will become a key feature of the Mizar style of formalizing mathematics.

## References

1. Bancerek, G.: On the Structure of Mizar Types. *Electronic Notes in Theoretical Computer Science* **85**(7) (2003).

[3] After running `trivdemo` there can occur `relprem` errors (when a whole proof is reduced to a straightforward justification) and new `trivdemo` errors (when a proof on a higher level is reduced), so that it has to be run several times to get complete results.

2. Kamareddine, F. and R. Nederpelt: A Refinement of de Bruijn's Formal Language of Mathematics. *Journal of Logic, Language and Information* **13**(3) (2004) 287–340.
3. Matuszewski, R. and P. Rudnicki: Mizar: The First 30 Years. *Mechanized Mathematics and Its Applications* **4**(1) (2005) 3–24.
4. Naumowicz, A.: Evaluating Prospective Built-in Elements of Computer Algebra in Mizar. *Studies in Logic, Grammar and Rhetoric* **10**(23) (2007) 191–200.
5. Rudnicki, P. and A. Trybulec: Mathematical Knowledge Management in Mizar. In Buchberger, B. and O. Caprotti (Eds.), *Electronic Proceedings of MKM 2001*. Available at `http://www.emis.de/proceedings/MKM2001/rudnicki.pdf`.
6. Schwarzweller, C.: Mizar Attributes: A Technique to Encode Mathematical Knowledge into Type Systems. *Studies in Logic, Grammar and Rhetoric* **10**(23) (2007) 387–400.
7. Trybulec, A.: Some Features of the Mizar Language. In Proceedings of ESPRIT Workshop, Torino (1993) available on-line at `http://mizar.uwb.edu.pl/project/trybulec93.ps`.
8. Wiedijk, F.: Checker. A compilation of e-mails written by Andrzej Trybulec, available on-line at `http://www.cs.ru.nl/~freek/mizar/by.ps.gz`.