Adjective Clustering in the MIZAR Type System

Adam Naumowicz

Institute of Informatics
University of Białystok, Poland
adamn@mizar.org

TYPES 2017, Budpest, May 29, 2017

About adjectives in MIZAR

- **Main motivation**: imitating as closely as possible the natural language of mathematics with its rich syntax.
- The idea was also present in de Bruijn's famous Mathematical Vernacular.
- The support for adjectives in the MIZAR language dates back to 1983/84, in a version called MIZAR HPF (with hidden parameters and functions).
- 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.
- When attributes were introduced in MIZAR, such changes were done semi-automatically to numerous predicates previously defined in the MIZAR library.

Adjectives/attributes terminology

- MIZAR "adjectives" are constructed using "attributes".
- They provide flexible type hierarchies in the collection of interdependent Mizar articles forming the Mizar Mathematical Library (MML).
- MIZAR adjectives are semantically variants of (dependent) predicates, but with
  - natural language based syntactic form,
  - built-in type inference automation.

Examples of attributes

- Without implicit parameters:

```
definition
  let R be Relation;
  attr R is well_founded means
  for Y being set st Y c= field R & Y <> {}
  ex a being set st a in Y & R-Seg a misses Y;
end;
```

- With an implicit parameter:

```
definition
  let n be Nat, X be set;
  attr X is n-at_most_dimensional means
  for x being set st x in X holds card x c= n+1;
end;
```

- With more implicit parameters:

```
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;
```

What is a "cluster of adjectives" in MIZAR jargon?

- 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.
- Grouping adjectives in clusters enables automation of some type inference rules (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),
  - allow formulating and automating relationships between adjectives (conditional registrations),
  - store adjectives that are always true for instantiations of terms with certain arguments (functorial registrations).

Examples of registrations

- Existential:

```
registration
  let n be Nat;
  cluster n-at_most_dimensional subset-closed non empty for set;
end;
```

- Conditional:

```
registration
  let n be Nat;
  cluster n-at_most_dimensional -> finite-membered for set;
end;
```

- Functorial (term):

```
registration
  let n be Nat;
  let X, Y be n-at_most_dimensional set;
  cluster X \/ Y -> n-at_most_dimensional;
end;
```

Biggest existential registration

```
registration
  let C be empty with_identities CategoryStr;
  let D be with_identities CategoryStr;
  cluster identity-preserving multiplicative antimultiplicative
  for Functor of C,D;
end;
```

MML Query

Mizar project
page generated with MMLQT (MML Query Transformation) tool

The biggest existential registrations (the query: list of exreg ordered by quantity of **cluster** reversed select 0-49)

1. CAT_6:exreg 15 includes 51 adjectives in registered cluster.
   registration
   let a_1 be empty with_identities CategoryStr;
   let a_2 be with_identities CategoryStr;
   cluster Relation-like non-empty empty-yielding (the carrier of a_1)-defined (the carrier of a_2)-valued empty trivial **non** proper epsilon-transitive epsilon-connected ordinal Sequence-like c=-linear natural zero non-zero without_zero Function-like one-to-one constant functional total quasi_total with_non-empty_elements finite finite-yielding finite-membered cardinal (0)-element FinSequence-like FinSubsequence-like FinSequence-membered complex ext-real **non** positive **non** negative real complex-valued ext-real-valued real-valued natural-valued increasing decreasing non-decreasing non-increasing identity-preserving multiplicative antimultiplicative for Element of bool [:the carrier of a_1,the carrier of a_2:];
   end;
2. CAT_6:exreg 17 includes 50 adjectives in registered cluster.
   registration
   let a_1 be empty with_identities CategoryStr;
   let a_2 be with_identities CategoryStr;
   cluster Relation-like non-empty empty-yielding (the carrier of a_1)-defined (the carrier of a_2)-valued empty trivial **non** proper epsilon-transitive epsilon-connected ordinal Sequence-like c=-linear natural zero non-zero without_zero Function-like one-to-one constant functional total quasi_total with_non-empty_elements finite finite-yielding finite-membered cardinal (0)-element FinSequence-like FinSubsequence-like FinSequence-membered complex ext-real **non** positive **non** negative real complex-valued ext-real-valued real-valued natural-valued increasing decreasing non-decreasing non-increasing covariant contravariant for Element of bool [:the carrier of a_1,the carrier of a_2:];

Adjective processing

- Original semantics
  - Mostly syntactic role, i.e. the Analyzer module automatically "rounded-up" the information from all available registrations to disambiguate used constructors and check their applicability.
  - The semantic role was restricted to processing only the type information for the terms explicitely stated in an inference.
    - Attributive statements as premises or conclusions were not "rounded-up".
    - The available automation did not take into account the potential of applying registrations to every element of a class of equal terms generated in the Equalizer module as a consequence of the equality calculus.
- Current optimized algorithm
  - "rounding-up" the, so called, "super clusters", i.e. clusters of adjectives collected from various representations of terms that happen to be aggregated in the same equality class as a consequence of equality processing.

Examples

With these two typical functorial registrations for integers encoded in the Mizar syntax:

```
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;
```

Mizar's Checker module can, for example, infer automatically the following statements as obvious for any `i`, `j`, `e` and `o` being integers:

```
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;
e is even & o is odd & i = j*e + o implies e+i is odd;
```

Some notes about MIZAR equality classes

- Equality classes are formed as a result of explicite equality statements and other language constructs: `set`, `reconsider` as well as e.g. built-in arithmetic.
- Equality classes 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.

Cluster matching algorithm

- Cluster matching reuses some of the data structures previously developed for the Unifier module.
  - An algebra of substitutions is used to contradict a given universal formula.
- The main difference is when joining instantiation lists:
  - in the Unifier the longer substitution is absorbed,
  - in the “super cluster” matching algorithm the longer substitution remains.

Cluster matching algorithm's basics

The calculus of (lists of) instantiations uses 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
- `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, two lattice-like constants:

- `TOP` which denotes a trivial substitution (no loci to be substituted, but all constants are matched)
- `BOTTOM` which is an empty list of substitutions (no match found).

`TOP` and `BOTTOM` have the usual lattice properties, e.g. are neutral with respect to the `MEET` and `JOIN` operations, respectively.

Cluster matching algorithm's main functions

- All these functions return as their result a (possibly empty) list of substitutions of classes for loci in the registration.
- For simplicity, we treat any class of terms `E` as a special kind of a term - one that satisfies the condition `E is CLASS`.

To check if a given class `E` matches a conditional registration `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
```

Cluster matching algorithm's main functions - ctd.

In the case of a functorial registration F, the matching function generates substitutions which match both the registered type and term of F.

- If a substitution is found, it can be used to extend the cluster of the equality class E.
- 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
```

Cluster matching algorithm's main functions - ctd.

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
```

Cluster matching algorithm's main functions - ctd.

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
```

Cluster matching algorithm's main functions - ctd.

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:

Cluster matching algorithm's main functions - ctd.

```
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
```

Cluster matching algorithm's main functions - ctd.

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 value, and their arguments match:

```
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
```

The cluster matching algorithm's main loop pseudo-code

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
```

Some interesting problems

- Rounding up adjectives is computationally expensive, but relatively simple if all the attributes are absolute, i.e. their only argument is the subject.
- In general, the subject may be defined with a type that has its own (explicit or implicit) arguments, and so the adjective may have more implicit arguments which complicates the "rounding-up" procedure.
- Efficient "rounding-up" clusters of adjectives with many arguments that can appear in clusters several times (but possibly with different arguments) is another non-trivial issue.