

Mizar Attributes: A Technique to Encode Mathematical Knowledge into Type Systems

Christoph Schwarzweller

Department of Computer Science
University of Gdańsk
ul. Wita Stwosza 57, 80-952 Gdańsk, Poland
schwarz@math.univ.gda.pl

Abstract. At first glance Mizar attributes look like unary predicates over mathematical objects enabling a more natural writing and reading. Attributes in Mizar, however, serve additional, more important purposes concerning typing of mathematical objects: Using attributes not only new (sub)types can be introduced, but also the user can characterize further relations between types and in this way make available existing notations for new objects. Thereby it should be stressed that these type relations can stand for elaborated mathematical theorems.

This paper describes the properties and benefits of Mizar attributes from a user's perspective. We comprehend the development of Mizar attributes, and give examples highlighting their use — essentially in the area of algebra. Concluding we discuss their impact on building mathematical repositories.

1 Introduction

After more than 30 years the Mizar system [6] is still widely-used and under active development. This is rather seldom for a mechanized reasoning system. The first presentation of Mizar in 1973 and the first write-up of 1975, that was published in 1977 [14], however, did not even mention proof checking as the main goal. At this stage Mizar basically was a language supposed to assist authors in the mathematical editing process. Therefore, though the language was meant to be understood and processed by a computer, great emphasis was placed on the linguistic component of the language. The point of reference here was the question how mathematicians write and present results in their papers. It is this fact, that from the very beginning an appealing formal language for mathematics was taken into consideration, that distinguishes Mizar from other systems in the area.

Later, of course, when proof checking for Mizar texts was developed, other items became equally important and influenced the further design of the Mizar language. The linguistic roots are still visible anyway and from our point of view provide the basis of Mizar's resistance against the course of time. This was again approved

by the success in the last couple of years, in which Mizar has evolved to one of the leading systems in the new area of mathematical knowledge management.

In this paper we appreciate the above mentioned development by focusing on Mizar attributes. In fact Mizar attributes have changed over the time from a poorly syntactical device to a powerful technique for defining and manipulating mathematical types in a user-friendly way — a topic that gains more and more importance, especially in mathematical knowledge management. We first review the development of Mizar attributes from the beginning and describe their present role in defining Mizar types. Then we illustrate in detail, how using attributes types can be introduced, refined and extended. The main point here is — in the spirit of mathematical knowledge management — the generation of a user-friendly environment, in which the provided knowledge can be reused, even in modified situations. In doing so, we concentrate on examples from the area of algebra. Finally, we discuss the impact of Mizar attributes on building large mathematical repositories.

2 Some History: Predicates, Attributes and Types

In the present version of the Mizar language predicates describe properties of mathematical objects, and so do attributes. The only difference is that while predicates may have any number of arguments, attributes must have exactly one. So, for example that a number x is even can be described by an attribute (or by a predicate), while that x is a divisor of y cannot. This was not always so. Although types were present from the very beginning, the first versions of Mizar employed predicates only. An attributive format for writing predicates was introduced in 1981 for the language Mizar-2 [7]. The reason was, as already indicated, a linguistical one. The new format enabled authors to write predicates in a more familiar form, such as

x is even
x is divisor of y.

Note, that the second predicate has two arguments, thus is not an attribute in today's sense, though here written in its attributive form.

This idea was further developed and in 1983 a number of extensions was realized in Mizar-HPF [7]. So, for example, more than one argument could be placed in front of the predicate's name. Also the keyword **is** could be replaced by **are**, if this seemed appropriate. In addition a more natural format for the negation of predicates using the keyword **not** was implemented. This more general attributive format lead to natural formulations such as

x, y are associated
x is not even.

Note, however, that all these phrases are just dealing with the natural writing of predicates and do not involve any kind of typing.

Mizar-4 [7] in 1986 saw the introduction of a special language construct, the so-called mode definition, for defining types (both with or without parameters),

that is still used today. Note that in this setting the second predicate from above should be rather defined as a mode `divisor` with one parameter. Also the concept of redefinitions was implemented, which allowed for redefining the type of a functor in case its arguments are more specific than in the original definition. For example the union of two finite sets could be redefined to be a finite set again. Note, however, that the type of finite sets at this stage was not constructed from the type set with the help of an additional adjective finite.

The development that started in Mizar-4 has been carried on and the present version of Mizar provides a very flexible type system in which attributes, or more precisely the adjectives constructed by attribute definitions, play a major role. Attribute definitions were separated from predicate definitions by the keyword `attr`, so that for example the definition of `even` now looks like

```

definition
let x be Integer;
attr x is even means 2 divides x;
end;

```

The negation of the defined attribute, now written using the keyword `non`, is automatically generated, so that the above definition introduces both adjectives `even` and `non even`. The negation of predicates in general, however, has not survived. The actual power of attributes for typing mathematical objects, however, rests on the rather obvious observation, that an attribute basically defines a subset of the type given in the definition. This allowed for the following extension of Mizar types. Radix types, the basic types given by mode definitions, can be equipped with adjectives in this way establishing a new subtype of the original radix type. Of course the adjective considered must be defined for this radix type. So, for example the adjective `even` generated by the above attribute definition together with the radix type `Integer` can be used to define the new type `even Integer` describing all objects of type `Integer`, that in addition fulfill the attribute `even`, in other words all even integers. In general (see [1] or [16]) Mizar types T have the form

$$T = \alpha_1 \dots \alpha_n R \text{ of } t_1, \dots t_k$$

where R is a radix type, the α_i are adjectives of R , the t_i are terms describing the parameters of type R and both n and k may be 0. Note that a type T_1 is a subtype of a type T_2 , if T_1 's radix type is a subtype of T_2 's radix type and T_2 's adjectives are a subset of T_1 's adjectives.

This is a very powerful scheme to describe types of mathematical objects. On the other hand in this setting it is very easy to define empty types such as for example the type

`empty infinite set.`

To prevent Mizar users from wasting time with non-existing mathematical objects, they are obliged to prove the existence of an object with the newly defined type. As a consequence empty types are not allowed in Mizar. The power of typing with adjectives, however, stems from the fact that adjectives can be also employed after the "main" definition of a type. More adjectives can be added to a type later and

even relations between set of adjectives can be stated (and again must be proven), in the sense that one set of adjectives logically implies another one. This effects Mizar's type system in that new subtype relations are introduced not comprised by the original definition. The consequence is that definitions, notations and theorems already elaborated for a mathematical type are now inherited to the new established subtype. We shall illustrate these techniques in detail in the next section with examples from the area of algebra.

3 Attributes and Registrations

In the following we show how Mizar attributes can be used to define algebraic types and subtypes (see also [12]). The main goal is to illustrate how in this context using attributes enables the reuse of notations and theorems already defined and proven for other types: By formulating (and proving) registrations the user can show that a mathematical object fulfills more adjectives than given in its definition, that is the object indeed have a more elaborated type than the one originally given. Even more, if the type is parameterized, registrations can be used to show that a more specific parameter type implies a more specific resulting type. In either case the registration "enriches" the Mizar checker in the sense that the checker automatically infers the new type given by the registration, which leads to the automated reuse of notations and theorems mentioned above. Note that using such registrations no original definition has to be modified in order to change or refine an object's type.

3.1 Existential Registrations

In defining algebraic types the radix type describes the carriers and operations of the algebraic structure. This is carried out in a so-called structure definition. The result is a new type describing mathematical objects with (at least) the components given in the definition. Thus, for example [4,15], the backbone structure for groups is given by the following structure definition `LoopStr` providing a carrier, a binary operation over the carrier and a distinguished element of the carrier. Note, that at this stage no further properties of the operation such as associativity or commutativity are required.

```
definition
struct (ZeroStr) LoopStr
  (# carrier -> set,
   add -> BinOp of the carrier,
   Zero -> Element of the carrier #);
end;
```

For a structure type Mizar automatically provides selectors allowing to access the individual components of the object, for example `the add of L` or `the Zero of L`, if `L` has type `LoopStr`. It is straightforward in Mizar to introduce the usual

mathematical notation for algebraic operations, in the case of groups $x + y$ for (the add of L). (x,y) and $0.L$ for the Zero of L. Note that, in contrast to $0.L$, the binary operation $x + y$ need not explicitly address the structure parameter L. The reason is that the parameter can be inferred from the type of the arguments x and y , which is **Element of the carrier of L** or shorter **Element of L**. Using these notations we can now formulate the usual properties of groups in a straightforward manner as attribute definitions. Note that the following attributes describe unary predicates over a **LoopStr L**.

```
definition
let L be non empty LoopStr;
attr L is add-associative means
  for x,y,z being Element of L holds (x + y) + z = x + (y + z);
attr L is right_zeroed means
  for x being Element of L holds x + 0.L = x;
attr L is right_complementable means
  for x being Element of L ex y being Element of L st x + y = 0.L;
end;
```

A group is now obviously a **LoopStr L**, that fulfills all three just defined attributes. This of course could have been also be defined using ordinary predicates such as `is_add-associative` etc. However, though describing group properties — and thus enabling to prove theorems for groups — predicates cannot be used to introduce the Mizar type **Group**. With the attributes from above we can define the attributed type **Group** in an existential registration as follows.

```
registration
cluster add-associative right_zeroed right_complementable
  (non empty LoopStr);
end;

definition
mode Group is add-associative right_zeroed right_complementable
  (non empty LoopStr);
end;
```

In an existential registration we summarize the attributes necessary to define a mathematical object. Note that the radix type **non empty LoopStr** to be refined must be explicitly given in the cluster. Because Mizar does not allow for empty types such a registration demands an existence proof, in which it has to be shown that there exists a mathematical object of radix type **non empty LoopStr** fulfilling the three mentioned adjectives. After that the type **Group** can be introduced as a synonym for the registered object. This, however, is not really necessary, because `add-associative right_zeroed right_complementable (non empty LoopStr)` is already a type for itself, so in fact **Group** is no more than an abbreviation.

For objects of type **Group** we can now formulate and prove theorems, that is for proving properties we can use exactly the definitions of the attributes constituting the type, and nothing more. Consequently, theorems proven this way are valid in all groups. For example, we can show that in groups the neutral element is unique:

```
theorem G1:  
for G being Group, x,y being Element of G  
holds x + y = x implies y = 0.G;
```

Now, groups are just the most general case. There are numerous more specific groups such as for example Abelian or solvable groups, which in a manner of speaking are refinements of the original group definition. The use of adjectives supports not only such refinements of mathematical types, but also guarantees that theorems for general groups like the example theorem G1 are automatically applicable to the refined types.

For example, to define Abelian groups, we simply introduce another attribute describing the additional property of commutativity and show — again in an existential registration — that there exists a group that fulfills this property. Afterwards the Mizar type `AbGroup` describing Abelian groups can be easily defined.

```
definition  
let L be non empty LoopStr;  
attr L is Abelian means  
  for x,y being Element of L holds x + y = y + x;  
end;
```

```
registration  
cluster Abelian Group;  
end;
```

```
definition  
mode AbGroup is Abelian Group;  
end;
```

The key point here is that `AbGroup` becomes a subtype of `Group`. This is due to the fact that both types share the same radix type and the adjectives of `Group` are a subset of the adjectives of `AbGroup`, or the other way round the type `AbGroup` widens to the type `Group`. As a consequence theorem G1 from above works also for the newly defined type `AbGroup`:

```
for G being AbGroup, x,y being Element of G  
holds x + y = x implies y = 0.G by G1;
```

Note that for introducing Abelian groups only one attribute definition (and an existence proof) were necessary, although afterwards Mizar provides the user of Abelian groups with a vast number of theorems for this type — all theorems proven for general groups.

There is, however, more to it than that [4]: We can also define new subtypes by extending the underlying structure, that is by refining the radix type. Again the mechanism of adjectives enables the natural reuse of already proven theorems. Fields, for example, are based on a structure that extends the one of groups by another binary operation and another element of the carrier. This can be directly expressed in Mizar by the following structure definition.

```
definition
struct (LoopStr, multLoopStr_0) doubleLoopStr
  (# carrier -> set,
   add, mult -> BinOp of the carrier,
   unity, Zero -> Element of the carrier #);
end;
```

The structure type `doubleLoopStr` in fact is a merge of two other ones: `LoopStr` and `multLoopStr` as can be seen in the definition. `doubleLoopStr` provides the components of both of them — further components can be introduced, but are not necessary in the case of fields. The main effect of this definition — besides the introduction of a field structure — is that the type `doubleLoopStr` becomes a subtype of both mentioned structure types `LoopStr` and `multLoopStr`. The algebraic type `Field` is now defined analogously to groups: Attributes describing field properties are introduced and after the existential registration we can introduce the corresponding Mizar type.

```
definition
mode Field is add-associative right_zeroed right_complementable
  Abelian commutative associative left_unital distributive
  Field-like non degenerated (non empty doubleLoopStr);
end;
```

Note, that the three attributes `add-associative` `right_complementable` and `right_zeroed` defined for `LoopStr` need not to be introduced for fields, that is for the type `doubleLoopStr`, again: Because the radix type `doubleLoopStr` widens to the type `LoopStr` they are available and can be reused.

Now every field is in particular a group with respect to addition, thus group theorems apply to fields also. As a result of the Mizar type system — and the definitions of groups and fields based on attributes — this is directly mirrored and feasible: The radix type of `Field` is a subtype of the radix type of `Group` and the adjectives of `Group` are a subset of the adjectives of `Field`. This implies that `Field` is a subtype of `Group`, and hence theorems for type `Group` are valid for type `Field` also. For our example theorem `G1` from above we get

```
for F being Field, x,y being Element of F
holds x + y = x implies y = 0.F by G1;
```

3.2 Functional Registrations

When defining algebraic structures functional registrations of attributes are used to introduce properties of concrete structures, such as integers or polynomials. Of course these properties can also be shown in theorems, proving for example that the integers constitute a ring. The main effect of such a functional registration, however, is again that the type of the mathematical object is extended. In this way not only theorems already proven for the refined type become available, but also definitions and notations introduced for mathematical objects of this type.

Consider the integers as an example [13]. In Mizar the integers are introduced as a `doubleLoopStr` providing the set of integers, addition and multiplication of integers as well as the numbers 0 and 1. The operations `addint` and `multint` are defined as binary operations over the set of integers `INT` and then identified with the corresponding components of `doubleLoopStr` in a function definition. Note that the function `In` changes the type of 0 and 1 to `Element of INT` — the proper type according to the definition of `doubleLoopStr`.

```

definition
func INT.Ring -> non empty doubleLoopStr equals
  doubleLoopStr(#INT,addint,multint,In(1,INT),In(0,INT)#);
end;

```

With this definition the following functional registration shows not only that `INT.Ring` fulfills the three attributes constituting a group, but also that `INT.Ring` is of type `Group` — remember that the type `doubleLoopStr` widens to the type `LoopStr`. Of course a (coherence) proof, that the three attributes are fulfilled by the object `INT.Ring`, is necessary.

```

registration
cluster INT.Ring -> add-associative right_zeroed right_complementable;
end;

```

One consequence of changing the type of `INT.Ring` is that our theorem `G1` from section 3.1 now applies to `INT.Ring` also. Note that the “group” parameter `G` from theorem `G1` has vanished, that is the following is a pure integer version of the general group theorem `G1`.

```

for x,y being Element of INT.Ring
holds x + y = x implies y = 0.(INT.Ring) by G1;

```

Another effect is that notations defined for the type `Group` are now available for `INT.Ring`, that is for integers. For example, the unary and binary operations `-` are defined for general groups the usual way: `-x` is the element of a group `G` such that `x + -x = 0.G` and `x - y` equals `x + (-y)`. Now `INT.Ring` having type `Group` there is no need to define these operations again for integers, they are just inherited from `Group`.

Of course one could argue that this is all obvious and that it could already be stated in the definition of `INT.Ring` that this object constitutes a ring. So there is no need to employ adjectives here. This is right, however, ignores the fact that later there can occur situations in which additional properties of an object such as `INT.Ring` are necessary to go on working. With the rather naive solution from above this would imply changing the definition in order to adopt the object’s type according to the additional properties — rather unpleasant, especially in a system with a large number of users. Using attributes, in contrast, we can easily include the additional properties into Mizar’s type system — without changing the original definition.

A typical example are Euclidean domains in which greatest common divisors can be easily computed using degree functions. The property of being `Euclidian`

is straightforwardly introduced as usual in an attribute definition. For objects of type `doubleLoopStr` fulfilling this attribute a degree function can be defined as follows. Note that Mizar expects an existence proof in such a definition, therefore the attribute `Euclidian` in the definition is essential.

```
definition
let E be Euclidian (non empty doubleLoopStr);
mode DegreeFunction of E -> Function of the carrier of E,NAT means
  (for a,b being Element of E st b <> 0.E holds
    (ex q,r being Element of E
      st (a = q * b + r & (r = 0.E or it.r < it.b))));
end;
```

Consequently `DegreeFunctions` exist only for objects of type `Euclidian` (non empty `doubleLoopStr`). The type of `INT.Ring` — using the above registrations — unfortunately does not widen to this type, although the radix types are identical. If we want to work with the integers as a Euclidean domain, we can, however, easily repair this “defect” — without changing the original definition of `INT.Ring`: Stating an additional registration for `INT.Ring`, in which we prove that `INT.Ring` fulfills the definition of `Euclidian`.

```
registration
cluster INT.Ring -> Euclidian;
end;
```

In this way, the fact that the ring of integers is Euclidean is not only proven, but also encoded into Mizar’s type system with the consequence that results for abstract Euclidean domains can be easily reused for integers, among others that there exists a `DegreeFunction` of `INT.Ring`.

The same can be done for algebraic structures with parameters — even if the additional property such as `Euclidian` is only valid if the parameter has additional properties also. Let us consider polynomial rings as an example [10]. Polynomial rings are defined analogously to the ring of integers as a function `Polynom-Ring(n,L)` with result type `doubleLoopStr`. Here the parameter `n` gives the number of indeterminates, the parameter `L` is the ring of coefficients. Then a number of functional registrations shows that `Polynom-Ring(n,L)` is of type `Ring`. Of course, if `L` is commutative, so is the resulting polynomial ring. Changing in this case the type of `Polynom-Ring(n,L)` to be in addition `commutative` is straightforward using a functional registration.

```
registration
let n be Ordinal,
    L be commutative Ring;
cluster Polynom-Ring(n,L) -> commutative;
end;
```

After this registration the type of the object `Polynom-Ring(n,L)` remains `Ring` if the parameter ring `L` is not commutative. If, however, `L` is commutative, that is

if L comes with the type `commutative Ring`, then the type of `Polynom-Ring(n,L)` inferred by Mizar is `commutative Ring` also.

We close this section with an example concerning again Euclidean rings. Polynomial rings are Euclidean, if the coefficient ring is in fact a field and the number of indeterminates is 1, so that in this case there exist degree functions for polynomials. The type `DegreeFunction of E`, however, only exists if E fulfills the attribute `Euclidian`. This can be easily expressed by the following registration.

```
registration
let F be Field;
cluster Polynom-Ring(1,F) -> Euclidian;
end;
```

Again we have two results: The mathematical statement is formally proven and the type of `Polynom-Ring(1,F)` for fields F is changed to `Euclidian Ring`, so that in this case the type `DegreeFunction of Polynom-Ring(1,F)` is available.

3.3 Conditional Registrations

The last kind of registration is used to deal with the types of classes of algebraic structures. This is of particular interest, if the adjectives stated in an object's definition imply other adjectives. In this case, conditional registrations allow to enrich the original type, so that the implied adjectives are automatically inferred. Of course again a (coherence) proof of the implication is necessary in Mizar.

For example it is obvious that in a commutative ring both left ideals and right ideals are in fact two-sided ideals. Proving these statements as theorems does not enrich the object's type: Notations defined for two-sided ideals are not available for left or right ideals in commutative rings — even after proving the theorem showing that they are (theoretically) available. Stating the proofs in the following conditional registration the first cluster changes the type `left-ideal (non empty Subset of R)` to `left-ideal right-ideal (non empty Subset of R)` if R is commutative. The second cluster does the analogous for right ideals.

```
registration
let R be commutative Ring;
cluster left-ideal -> right-ideal (non empty Subset of R);
cluster right-ideal -> left-ideal (non empty Subset of R);
end;
```

Thus after this registration in commutative rings objects of type `left-` or `right-ideal` automatically have type `(two-sided) ideal` also. Note that to establish a subset of a commutative ring as an ideal it is then sufficient to show that the subset is a left (or a right) ideal.

For a more involved example we return to groups as described in section 3.1. There we defined groups G using the attribute `right_zeroed` describing the axiom $x + 0.G = x$. Consequently the following statement is automatically accepted by the Mizar checker.

```
for G being Group holds G is right_zeroed;
```

Now, in a group G we also have $0.G + x = x$, that is G is `left_zeroed` also. The corresponding statement, however, is not automatically accepted by the Mizar checker, because the type `Group` does not include the corresponding attribute `left_zeroed`. Of course, one could add this property to the group definition. But this would imply, that in the definition is more stated than necessary, rather un-aesthetic from a mathematical point of view. The solution again is a conditional registration showing that the attributes used to define the type `Group` imply the attribute `left_zeroed`. Note that the antecedent of the cluster is empty, because all necessary attributes are given by the type `Group`. After the registration the Mizar checker automatically accepts the fact that an object of type `Group` fulfills the attribute `left_zeroed`.

```
registration
cluster -> left_zeroed Group;
end;

for G being Group holds G is left_zeroed;
```

The same technique can again be applied to parameterized types. We illustrate this by introducing subgroups. A subgroup is a subset of a group that is itself a group. This can be easily described by the following mode definition. Note that it is necessary to explicitly express that the addition and the neutral element are inherited from the given group G .

```
definition let G be Group;
mode Subgroup of G -> Group means
  the carrier of it c= the carrier of G &
  the add of it = (the add of G)||the carrier of it &
  the Zero of it = the Zero of G;
end;
```

Now, as the above definition states, a subgroup of a group is in particular of type `Group`. Hence, because — according to section 3.1 — Abelian groups are in particular groups, the type `Subgroup of G` where G is an Abelian group exists and the following statement about Abelian groups is obvious for the Mizar checker.

```
for G being AbGroup, H being Subgroup of G holds H is add-associative;
```

Of course a subgroup of an Abelian group is again Abelian. The corresponding statement, however, is not automatically accepted by the Mizar checker. The reason is obvious: According to our definition of subgroups the type of a subgroup of G is `Group`, even if G is of type `Abelian Group`. Therefore the Mizar checker cannot infer, that in this case the attribute `Abelian` is also fulfilled. In the following conditional registration we restrict the type of the parameter G to `Abelian Group` and show that then each object of type `Subgroup of G` fulfills the attribute `Abelian`. As usual this registration also enriches the Mizar type checker, so that the type for subgroups of Abelian groups G is changed into `Abelian Subgroup of G`.

```
registration
let G be Abelian Group;
cluster -> Abelian Subgroup of G;
end;
```

As a consequence the fact that subgroups of Abelian subgroups are Abelian is automatically accepted. Even more, because the type `Field` is a subtype of `AbGroup` — note that the defining cluster for `Field` in section 3.1 includes the attribute `Abelian` — this is also true for this type; and in fact for every other algebraic type widening to the type `AbGroup` such as for example `Ring`. Hence all the following statements are now obvious for the Mizar checker.

```
for G being AbGroup, H being Subgroup of G holds H is Abelian;
```

```
for F being Field, H being Subgroup of F holds H is Abelian;
```

```
for R being Ring, H being Subgroup of R holds H is Abelian;
```

We close this section with a “type version” of a famous result by J. Wedderburn. Skew fields or division rings [8] differ from fields only in that their multiplication is not required to be commutative. Finite skew fields however — as J. Wedderburn showed — are always commutative. This statement can be formulated as a conditional registration as follows.

```
registration
cluster finite -> commutative Skew-Field;
end;
```

Again this fact could just as well be stated as a theorem. But note that the registration in fact changes the type `finite Skew-Field` to `Field`, because now all attributes of the `Field`’s definition are fulfilled. Hence, due to the last registration in section 3.2, polynomial rings over finite skew fields are Euclidean. As a consequence the type `DegreeFunction of Polynom-Ring(1,F)` for finite skew fields `F` is automatically available, which would not be true if Wedderburn’s theorem would have been stated as an “ordinary” Mizar theorem.

4 Conclusions

As we have seen Mizar attributes have developed from a pure linguistic phrase into a powerful technique for defining and manipulating types. Though basically all the mathematical definitions and statements in section 3 can be expressed without using attributes and registrations, there are good reasons not to do so, notably in mathematical repositories.

Firstly, we consider the use of attributes in mathematical repositories user-friendly. Attributes allow to enrich algebraic structures with additional properties without changing the original definition. In this way attributes also support the reuse of already proven theorems: Refining a type by adding additional attributes

does not exclude the use of theorems formulated for the original type. In addition new type relations can be established. Secondly, typing with attributes shortens the number of theorems to be stored in a mathematical repository. Note that the example theorem from section 3.1 need to be stored only once for the type **Group**, although we presented versions for four different algebraic structures; a number that could be easily enlarged. Thirdly, attributes allow to handle algebraic structures in a mechanized reasoning system similarly to the way mathematicians do. Mathematicians change their view on an algebraic structure if this seems convenient. For example group theorems are applied to fields without thinking about the type coercion that actually takes place. As we have seen Mizar attributes not only enable such a dealing with algebraic structures but in addition allow for automating even ambitious theorems in a mathematical repository.

We believe that especially the linguistic motivation for introducing attributes to the Mizar language permitted in later years the development of a typing technique that is extremely well-suited for the development of mathematical repositories. So, we wait and hope for further “linguistical inspiration” from Mizar. In this spirit

Andrzeju, wszystkiego najlepszego!

References

1. G. Bancerek, *On the Structure of Mizar Types*; Electronic Notes in Theoretical Computer Science, vol. 85(7), Elsevier, 2003.
2. N.G. de Bruijn, *The Mathematical Vernacular, a language for mathematics with typed sets*; in P. Dybjer et al. (eds.), Proc. of the Workshop on Programming Languages, Marstrand, Sweden, 1987.
3. J.H. Davenport, *MKM from book to computer: a case study*; in: A. Asperti, B. Buchberger, and J. Davenport (eds.), Proc. of MKM 2003, Lecture Notes in Computer Science 2594, pp. 17–29, 2003.
4. E. Kusak, W. Leończuk and M. Muzalewski, *Abelian Groups, Fields and Vector Spaces*; Formalized Mathematics, vol. 1(2), pp. 335–342, 1990.
5. F. Kamareddine and R. Nederpelt, *A refinement of de Bruijn’s formal language of mathematics*; Journal of Logic, Language and Information, vol. 13(3), pp. 287–340, 2004.
6. The Mizar Homepage, www.mizar.org.
7. R. Matuszewski and P. Rudnicki, *Mizar: The First 30 Years*; Mechanized Mathematics and Its Applications, vol. 4(1), pp. 3–24, 2005. <http://mizar.org/people/romat/MatRud2005.pdf>
8. M. Muzalewski, *Construction of Rings and Left-, Right-, and Bi-Modules over a Ring*; Formalized Mathematics, vol. 2(1), pp. 3–11, 1991.
9. P. Rudnicki and A. Trybulec, *Mathematical Knowledge Management in Mizar*; in: B. Buchberger, O. Caprotti (eds.), Proc. of MKM 2001, Linz, Austria, 2001.
10. P. Rudnicki and A. Trybulec, *Multivariate Polynomials with Arbitrary Numbers of Variables*; Formalized Mathematics, vol. 9(1), pp. 95–110, 2001.
11. P. Rudnicki and A. Trybulec, *On the integrity of a repository of formalized mathematics*; in: A. Asperti, B. Buchberger, and J. Davenport (eds.), Proc. of MKM 2003, Lecture Notes in Computer Science 2594, pp. 162–174, 2003.

12. P. Rudnicki, A. Trybulec and C. Schwarzweller, *Commutative Algebra in the Mizar System*; Journal of Symbolic Computation, vol. 32(1/2), pp. 143-169, 2001.
13. C. Schwarzweller, *The Ring of Integers, Euclidean Rings and Modulo Integers*; Formalized Mathematics, vol. 8(1), pp. 29-34, 1999.
14. A. Trybulec, *Informationslogische Sprache Mizar*; Dokumentation-Information, Heft 33, Ilmenau, 1977.
15. W.A. Trybulec, *Vectors in Real Linear Space*; Formalized Mathematics, vol. 1(2), pp. 291-296, 1990.
16. F. Wiedijk, *Writing a Mizar Article in Nine Easy Steps*; available from <http://www.mizar.org/project/bibliography.html>.