Defining Power Series and Polynomials in Mizar

Piotr Rudnicki*
University of Alberta, Canada
email: piotr@cs.ualberta.ca

Christoph Schwarzweller
University of Tübingen, Germany
email: schwarzw@informatik.uni-tuebingen.de

Andrzej Trybulec†
University of Białystok, Poland
email: trybulec@math.uwb.edu.pl

14th September 2000

Abstract. *We report on the construction of formal multivariate power series and polynomials in the Mizar system. First, we present how the algebraic structures are handled and how we inherited the past developments from the Mizar library. The Mizar library evolves and past contributions are revised and (usually) generalized. Our work on formal power series caused a number of such revisions. It seems that revising past developments with an intent to generalize them is a necessity when building a data base of formalized mathematics. And this poses a question: how much generalization is best?*

*Supported by NSERC Grant OGP9207.
†Supported by NSERC Grant OGP9207 and NATO CRG 951368.

1 Introduction

Mathematics, especially algebra, uses dozens of structures; groups, rings, vector spaces, to name few of the most basic ones. These structures are closely connected to each other giving raise to inheritance. For example, each ring is a group with respect to its addition and hence every theorem about groups trivially holds for rings also. Furthermore there is a trend towards introducing more general structures: semi-rings as a generalization of rings, modules as a generalization of vector spaces, etc. Again, theorems about a structure are trivially true for any structure derived from it. The derived structure inherits everything from its ancestors.

In mechanized proof-checking systems the issues of inheritance have to be made explicit. It is by far non trivial to build a proof-checker for which theorems for groups apply also to rings. Generalizations, as mentioned above, may result in building a sizeable graph of inheritance and only extensive practice can say how good is a particular solution. The issue is further complicated by inertia induced through the existing developments in a proof-checking environment. On the one hand, one would like to inherit as much as possible from the past, on the other hand one wants to change the past if it turns out to be inconvenient for the task at hand. And the task at hand is usually too big to start everything from scratch.

In this paper we describe the construction of formal multivariate power series and polynomials in the Mizar system [4], during which we had to deal with these problems. We discuss the tools Mizar offers to build algebraic structures; tools, we believe, providing a flexible mechanism allowing the kind of inheritance omnipresent in mathematics. Generalization is a more complex task. Of course one can easily derive rings from semi-rings; however, there is a challenge when the rings have been already introduced in the past and one aims at introducing semi-rings. The question is what to do with the theorems about rings already proven and stored in a library. A number of them will hold for semi-rings also. Stating and proving them again would not only be a tedious job but would blow up the library. Alternatively, the library has to be revised as a whole. We discuss some issues of generalizations and library revisions.

2 Defining Algebraic Domains in Mizar

The Mizar construction of formal multivariate polynomials aimed at defining the ring of polynomials over a minimal algebraic domain which would permit such a construction. The definition of an algebraic domain is founded on a structure mode providing the primitive notions, and next,

the axioms for a specific class of structures are defined as properties of the underlying structure mode.

In our case, the structure mode of interest is `doubleLoopStr`, defined in [3]. Figure 1 illustrates the relationship of `doubleLoopStr` to other structure modes[1]. The bottom definition introduces the following constructors:

- The structure mode `doubleLoopStr`, that may be used to qualify variables, e.g. `let S be doubleLoopStr` or form predicates, e.g. `T is doubleLoopStr`. However, `T` for which the above holds, may have other fields besides those listed in the definition, if the type of `T` is derived from `doubleLoopStr`.

- The attribute `strict` which when used as `strict doubleLoopStr` gives the type of structures that have no additional fields besides the ones mentioned in the definition. The attribute symbol `strict` is heavily overloaded as every definition of a structure mode defines a new attribute denoted by this symbol.

- The aggregate functor that is used to construct aggregates of the form `doubleLoopStr(#c,a,m,u,z#)`, where `c` is a set, `a` and `m` binary operations on `c`, `u` and `z` two fixed elements of `c`. Structures denoted by aggregates are `strict`.

- The forgetful functor which when used as `the doubleLoopStr of S` creates a strict structure from `S` (provided `S` has a type widening to `doubleLoopStr`). This functor denotes the aggregate:

```
doubleLoopStr (# the carrier of S,
                 the add of S, the mult of S,
                 the unity of S, the Zero of S #)
```

  If `S` is `strict doubleLoopStr`, then `S` = `the doubleLoopStr of S`.

The mode `doubleLoopStr` is derived from `LoopStr` and `multLoopStr_0`. This means that type `doubleLoopStr` widens to both, or in other words is a subtype of both `LoopStr` and `multLoopStr_0`.

Typically, a structure definition introduces also some selector functors to access its fields. The selector functors are introduced in the first structure definition in which the selector is used. The structure mode `1-sorted` defines the selector functor `the carrier of`. It may be used for any `1-sorted` structure, e.g. `ZeroStr`, `LoopStr`, `doubleLoopStr`. The selector functor `the Zero of` is introduced by `ZeroStr` and the selector functor `the add of` by LoopStr. In the case of `multLoopStr_0` no new selectors are

[1] See [1], [8] and [3] to learn more about these structures.

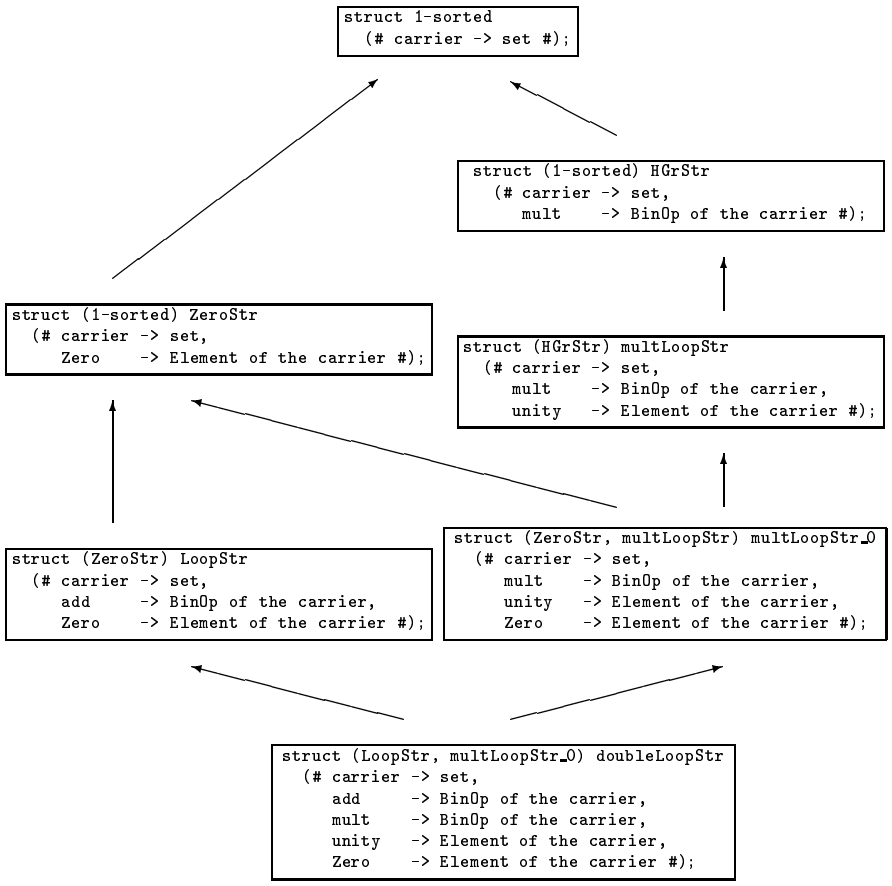

Figure 1. Derivation of `doubleLoopStr`

introduced, `mult` and `unity` are inherited from `HGrStr` and `multLoopStr`, respectively. In the case of `doubleLoopStr` also all selector functors are inherited.

`ZeroStr` is a common ancestor of `LoopStr` and `multLoopStr_0`. In this way we ensure that `carrier` and `Zero` are the same in both. The definition of `ZeroStr` introduces `Zero` as a new selector, `carrier` is inherited from `1-sorted` that is common ancestor for most structures in the Mizar library (MML).

If `S` is defined to satisfy `S = doubleLoopStr(#c,a,m,u,z#)` then

```
  the 1-sorted of S = 1-sorted(#c#)
  the ZeroStr of S = ZeroStr(#c,z#)
  the LoopStr of S = LoopStr(#c,a,z#)
  the multLoopStr_0 of S = multLoopStr_0(#c,m,u,z#)
  the doubleLoopStr of S = doubleLoopStr(#c,a,m,u,z#)
```

that is `the doubleLoopStr of S = S`.

The order of selectors in a structure definition serves syntactic purposes only and it can be chosen arbitrarily (with obvious restriction that a selector s_1 that occurs in the type of a selector s_2 must be put before s_2). The structures in Mizar are not tuples but rather partial functions on selectors and selectors must not be identified with just a place in the aggregate functor.

A structure mode defines a backbone on which algebraic domains are built. The desired properties of an algebraic domain are introduced usually one at a time through defining appropriate attributes. For example, associativity of addition is formulated ([8]) as:

```
definition
let S be non empty LoopStr;
attr S is add-associative means
  for a,b,c being Element of the carrier of S
  holds (a + b) + c = a + (b + c);
end;
```

(`a + b` is a shorter notation for `(the add of S).[a,b]`; this notation is usually defined right after the selector functor is introduced, see [8].) The attribute `add-associative` is defined for the structure mode `LoopStr`, in which the selector `add` is introduced. Through inheritance—the mode `doubleLoopStr` widens to the mode `LoopStr` as the latter is an ancestor of the former—the attribute is available for objects of mode `doubleLoopStr`.

Using separate attributes, various properties of algebraic domains are defined. These attributes can then be combined into clusters:

```
definition
cluster add-associative right_zeroed right_complementable
        Abelian commutative associative left_unital
        right_unital distributive Field-like non degenerated
        (non empty doubleLoopStr);
existence
  proof
    Demonstrate the existence of an object with all listed attributes.
  end;
end;
```

The attributes in the cluster above were introduced for various structure modes, all inherited by `doubleLoopStr`. For example, `empty` is defined for `1-sorted`, `Abelian` for `LoopStr`, `commutative` for `HGrStr`. The attribute `distributive` is defined for `doubleLoopStr` as it could not have been defined earlier.

The existence proof in the cluster definition is necessary to avoid empty modes that are not allowed in Mizar. Once we have proven the existence of an object with a cluster of attributes, we can introduce a mode of the desired algebraic domain:

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

The mode `Field` is an abbreviation for a `doubleLoopStr` having the attributes given in its definition. Note that through inheritance the definition of `Field` just combines various notions; most of them exist on their own merit. A definition of a `Ring` could share the same backbone structure with `Field` and its attributes could be a subset of `Field`'s attributes. Therefore each theorem about a `Ring` would be applicable to a `Field`.

Conditional clusters is another Mizar mechanism through which we obtain reuse of theorems. A conditional cluster expresses the fact that a Mizar object that enjoys some attributes also enjoys another ones. For example, the rather trivial fact that a commutative binary operator with a right zero also possesses a left zero, can be expressed as follows:

```
definition
cluster Abelian right_zeroed -> left_zeroed (non empty LoopStr);
coherence
  proof
    Demonstrate that the desired implication holds.
  end;
end;
```

Once the above conditional cluster has been registered, predicates and functions defined for `left_zeroed LoopStr` are now also available for all other objects whose type widens to `Abelian right_zeroed LoopStr`. Also, theorems proven for `left_zeroed LoopStr` are now applicable to `Abelian right_zeroed LoopStr` and all other types widening to it. Mizar checker

tacitly processes all available conditional clusters and they are not explicitly referenced.

3 Formal Multivariate Power Series and Polynomials

The construction of formal power series and polynomials is presented in [5]. We build power series as functions from power products into a structure `L` of coefficients. A power product itself is a function, called `bag`, from a given set of variables into the natural numbers.

Variables are elements of an arbitrary set `X`. When we need the variables to be ordered, we use ordinals as `X` but we prefer to be as general as possible. A bag over a set of variables `X` is defined in terms of the concept of `ManySortedSet` [7].

```
definition
let X be set;
mode bag of X is
  natural-yielding finite-support ManySortedSet of X;
end;
```

The attribute `natural-yielding` means that the values of a bag are natural numbers, whereas `finite-support` describes the property of a function as having only finitely many values not equal to zero. The set of all bags of `X` is then defined and named `Bags X`.

Several operations on bags are defined, for example addition (`b1 + b2`) used for multiplying power products and restricted subtraction (`b1 -' b2`) used for dividing power products. Also, we introduced the usual order on power products and the concept of their divisibility.

Given a structure `S`, a formal power series over `S` with the variables in `X` assigns to each power product over `X` a coefficient, an element of `S`. Consequently a `Series of S,X` is a function from `Bags X` into `S`:

```
definition
let X be set, S be 1-sorted;
mode Series of X,S -> Function of (Bags X),S means
  not contradiction;
end;
```

Note that nothing is required from the structure `S`, in particular no addition over `S` has to be available. These assumptions are introduced later when necessary to ensure additional properties of series. For example, defining addition of series requires addition of the elements of `S`, hence is defined for `LoopStr`:

```
definition
let n be set, L be right_zeroed (non empty LoopStr),
    p,q be Series of n,L;
func p + q -> Series of n,L means
  for x being bag of n holds it.x = p.x + q.x;
end;
```

(Note: the attribute `right_zeroed` is not really needed and will be probably eliminated when the library is revised.)

Definition of multiplication required a bit more work: `p*q` on a bag `b` is obtained by considering all decompositions of `b` into bags `b1` and `b2` such that `b = b1 + b2`. This is done with the helper functor `decomp` which gives the finite sequence of decompositions of `b` ordered in increasing order of the first component. For this we required that the variables are identified with a certain ordinal:

```
definition
let n be Ordinal,
    L be add-associative right_complementable
         right_zeroed (non empty doubleLoopStr),
    p,q be Series of n, L;
func p*q -> Series of n, L means
  for b being bag of n
  ex s being FinSequence of the carrier of L
  st it.b = Σ s &
     len s = len decomp b &
     for k being Nat st k ∈ dom s
     ex b1, b2 being bag of n st π(decomp b,k) = <*b1, b2*> &
                                 π(s,k) = p.b1 · q.b2;
end;
```

(We would like to mention that proving the associativity of this multiplication presented a technical challenge.)

We also defined the operators `p - q` and `-p` with the obvious meaning as well as the zero series and the unit series denoted by `0_(n,L)` and `1_(n,L)`.

Polynomials are a special case of formal power series; they are the series having only finitely many power products with non-zero coefficients, that is series with a finite support (written `finite-Support` to distinguish from `finite-support`, introduced earlier). Due to this restriction the underlying structure `L` must have a zero, hence be a `ZeroStr`.

```
definition
let n be Ordinal, L be non empty ZeroStr;
mode Polynomial of n,L is finite-Support Series of n,L
```

```
end;
```

All the functors defined for series and resulting in series can be applied to polynomials, however, the types of these functors are series and not polynomials. We have to explicitly state when performing operations on polynomials we obtain polynomials, that is that the resulting series has finite support. This problem is solved by employing functorial clusters, in which exactly this is stated (and proved), for example:

```
definition
let n be Ordinal, L be right_zeroed (non empty LoopStr),
    p,q be Polynomial of n,L;
cluster p + q -> finite-Support;
coherence
  proof
   Show that the result of adding two Polynomials has finite-Support.
  end
end;
```

Putting it all together we get the ring of polynomials over a structure `L` as a `doubleLoopStr` in which the single components are identified with the corresponding just defined operators. Note that the underlying structure `L` is not a full commutative ring. We only used attributes necessary to ensure that the operators for polynomials again result in a polynomial.

```
definition
let n be Ordinal,
    L be right_zeroed add-associative right_complementable unital
         distributive non trivial (non empty doubleLoopStr);
func Polynom-Ring(n,L) -> strict non empty doubleLoopStr means
  (for x being set
   holds x ∈ the carrier of it iff x is Polynomial of n,L) &
  (for x,y being Element of it, p,q being Polynomial of n,L
   st x = p & y = q holds x + y = p + q) &
  (for x,y being Element of it, p,q being Polynomial of n,L
   st x = p & y = q holds x · y = p * q) &
  0.it = 0_(n,L) &
  1_ it = 1_(n,L);
end;
```

So far we only defined an instance of a `doubleLoopStr`; nothing is said about the usual algebraic properties of a polynomial ring. To constitute `Polynom-Ring(n,L)` as a ring, the necessary attributes are introduced in cluster registrations. For some of the attributes, additional properties of `L`

were necessary. For example, it turned out that in order to prove the commutativity of multiplication, we also needed the addition of polynomials to be commutative.

```
definition
let n be Ordinal,
    L be Abelian add-associative right_zeroed
         right_complementable commutative unital distributive
         non trivial (non empty LoopStr);
cluster Polynom-Ring(n,L) -> commutative;
end;
```

Finally, to prove distributivity of `Polynom-Ring(n,L)` we had to use attributes implying that `L` is a ring with a unit, but not a commutative one.

```
definition
let n be Ordinal,
    L be right_zeroed Abelian add-associative
         right_complementable unital distributive associative
         non trivial (non empty doubleLoopStr);
cluster Polynom-Ring (n, L) -> unital right-distributive;
end;
```

4 Evaluating Multivariate Polynomials

The next natural step is to define the evaluation of polynomials in the underlying structure `L` [6]. To define the evaluation as a function from the ring of polynomials over `L` into `L`, it is not necessary for `L` to be a ring. But in order to prove that the evaluation of polynomials is a homomorphism, further properties of `L` are necessary, namely that `L` is a non trivial commutative ring with 1.

First, we resolve the problem of evaluating a power product `b` which is a `bag of n`.

```
definition
let n be Ordinal,
    b be bag of n,
    L be unital non trivial (non empty doubleLoopStr),
    x be Function of n,L;
func eval(b,x) -> Element of the carrier of L means
  ex y being FinSequence of the carrier of L
  st len y = len SgmX(RelIncl n, support b) &
     it = Π y &
     for i being Nat st 1 <= i & i <= len y holds
```

```
        y|.i = power(L).((x·SgmX(RelIncl n, support b))|.i,
                         (b·SgmX(RelIncl n, support b))|.i);
end;
```

We use a helper function `x` evaluating the variables `n` into `L`. To get the interesting part of `x`, that is an evaluation of the variables occurring with non-zero exponents in `b`, the functor `SgmX` is employed. This functor takes a finite set—here the support of a bag—and a linear order for the set and returns a finite sequence in which the elements of the set occur in increasing order. This sequence is then composed with `x` to get the finite sequence of values and with `b` to get the corresponding finite sequence of exponents. The exponentiation is then performed pointwise yielding a finite sequence `y` of elements of `L`. The result of the evaluation of `b` with respect to `x` is the product of the values of `y`. We get this product using the functor Π which takes a finite sequence (over a structure allowing for elements occurring in this sequence [9]).

The structure `L` has to meet two requirements: the existence of a unity and that it is not trivial (has at least two elements). However, in order to prove that the evaluation respects multiplication of power products, that is to prove the rather obvious fact

$$\texttt{eval(b1+b2,x)} = \texttt{eval(b1,x)} \cdot \texttt{eval(b2,x)}$$

it turned out that `L` has to provide a commutative multiplication with a (left and right) unity. As this property is necessary to prove multiplicativity of evaluation, it follows that the evaluation of polynomials is a homomorphism only if the underlying structure `L` is a commutative ring with 1.

The definition of the evaluation of a polynomial is defined in analogous way to the evaluation of power products:

```
definition
let n be Ordinal,
    L be right_zeroed add-associative right_complementable unital
         distributive non trivial (non empty doubleLoopStr),
    p be Polynomial of n,L,
    x be Function of n,L;
func eval(p,x) -> Element of the carrier of L means
  ex y being FinSequence of the carrier of L
  st len y = len SgmX(BagOrder n, Support p) &
     it = Σ y &
     for i being Nat st 1 <= i & i <= len y holds
        y|.i = (p·SgmX(BagOrder n, Support p))|.i ·
               eval(((SgmX(BagOrder n, Support p))|.i),x);
end;
```

The next goal was to prove that the functor `eval` is a homomorphism from the polynomial ring over `L` into `L`. To do so we introduced a functor `Polynom-Ring(n,L,x)` taking an ordinal number, a structure `L` and variable evaluation function `x` as parameters, assigning to each polynomial `p` the value of `eval(p,x)`.

```
definition
let n be Ordinal,
    L be right_zeroed add-associative right_complementable unital
         distributive non trivial (non empty doubleLoopStr),
    x be Function of n, L;
func Polynom-Evaluation(n,L,x) -> map of Polynom-Ring(n,L),L
   means for p being Polynomial of n,L holds it.p = eval(p,x);
end;
```

Proving the properties of a homomorphism required additional assumptions concerning `L`, in particular, to prove that the evaluation of polynomials is compatible with the multiplication of polynomials we had to assume for the first time, that the underlying structure `L` is indeed a commutative ring with 1. Thus we ended up with the following

```
definition
let n be Ordinal,
    L be right_zeroed add-associative right_complementable
         Abelian well-unital distributive non trivial
         commutative associative (non empty doubleLoopStr),
    x be Function of n,L;
cluster Polynom-Evaluation(n,L,x) -> RingHomomorphism;
end;
```

5 Library Revisions

When starting to define polynomials, we wanted to keep the number of new definitions as small as possible. Hence we examined the Mizar Mathematical Library (MML) to find concepts we could use for our task. Several problems occurred.

We found out (not for the first time) that many basic theorems were missing. For example, the functor Σ sums up elements of a finite sequence; it is clear that if all but one particular element equal zero, the sum in fact is this element. This theorem had not been proven before.

Some concepts were introduced for a too specific structure, hence not general enough to use them in our case. For example, the functor `power`, for exponentiation with natural numbers, was defined for groups, whereas

we wanted to use it in a structure providing only unity. Of course one can define the functor again for the more general case, but this does not seem appropriate in a library. The solution is to revise the MML, that means generalizing the original definition and reformulating the theorems concerning this concept.[2]

On the one hand this problem seems natural. If one is writing an article about for example groups in which one needs a functor—and one does not find it in the MML—one simply defines it. And why should one think about more general solutions, if it works well for the theorems intended to prove? In addition, it is rather hard, if even possible, to estimate how general a definition should be in order to provide optimal benefit for future users of MML.

On the other hand, while proving theorems about the new concept, one usually observes which properties of the underlying structure are necessary to prove it and which are not. The correctness proof of the `power` functor, for example, definitely did not use the properties of a group. The same holds for defining formal power series and polynomials: we first did (and finished) this job for polynomials with a finite number of variables only, before we realized that we already had developed all the machinery for constructing power series in arbitrary number of variables.

Another point connected with this problem is that sometimes it may be better to be not as general as possible. For example, although one can build the theory of polynomials in one variable out of our approach, by using `Polynomial-Ring(1,R)`, this seems not to be the best solution. Doing so would require `R` to be a commutative and associative ring, just because these properties were necessary to prove multiplicativity of the evaluation in the general case. But it seems that for polynomials with one variable only this property can be established with weaker assumptions on `R`.[3] So the question remains: How much generalization is best?

6 Conclusions and Further Work

In this paper we described the construction of formal multivariate power series and polynomials in the Mizar system. The main concern was to present the possibilities Mizar offers to build algebraic structures. Although these possibilities are quite elegant and include inheritance in the usual mathematical style, library revisions were necessary during our work.

[2] Sometimes it turns out that the proof of a theorem actually does not use all properties of the structure it is about.

[3] At the moment the theory of polynomials in one variable is developed separately in order to check this.

We plan to go on with our work on polynomials, among other goals is getting more insight into the way of dealing with inheritance in algebraic structures. We want to check how applications of our general theory of polynomials can be done. One of the goals is to develop a theory of polynomials over finite fields—as used in coding theory—thus to restrict the underlying structure of a polynomial ring. Also the theory of one-variable polynomials will be constructed separately, although it could be build using our approach. This may serve as a case study concerning the question how much generalization is best.

Another plan is to work towards the theory of Groebner bases and Buchberger-like algorithms. For that it is necessary to develop the theory of ideals first. This seems to be an algebraic topic with many applications and hence could also contribute to solve the problems we discussed here.

Bibliography

[1] Association of Mizar Users, Library Committee, *Preliminaries to Structures*. Available on WWW:
`http://mizar.org/JFM/Addenda/struct_0.abs.html`.

[2] Grzegorz Bancerek and Krzysztof Hryniewiecki, *Segments of Natural Numbers and Finite Sequences. Formalized Mathematics*, 1(**1**):107–114, 1990. Available on WWW:
`http://mizar.org/JFM/ Vol1/finseq_1.abs.html`.

[3] Eugeniusz Kusak, Wojciech Leończuk and Michał Muzalewski, *Abelian Groups, Fields and Vector Spaces. Formalized Mathematics*, 1(**2**):335–342, 1990. Available on WWW:
`http://mizar.org/JFM/Vol1/vectsp_1.abs.html`.

[4] Piotr Rudnicki and Andrzej Trybulec. On Equivalents of Well-foundedness. An experiment in Mizar. *Journal of Automated Reasoning*, 23:197–234, 1999.

[5] Piotr Rudnicki and Andrzej Trybulec, *Multivariate Polynomials with arbitrary Number of Variables. Formalized Mathematics*, 8(**1**):317–332, 1999. Available on WWW:
`http://mizar.org/JFM/Vol11/polynom1.abs.html`.

[6] Christoph Schwarzweller and Andrzej Trybulec, *Evaluation of Multivariate Polynomials.* To appear in *Formalized Mathematics*, 2000. Available on WWW:
`http://mizar.org/JFM/Vol12/polynom2.abs.html`.

[7] Andrzej Trybulec, *Many-sorted sets. Formalized Mathematics*, 4(**1**):15–22, 1993. Available on WWW:
`http://mizar.org/JFM/Vol5/pboole.abs.html`.

[8] Wojciech A. Trybulec, *Vectors in Real Linear Space. Formalized Mathematics*, 1(**2**):291–296, 1990. Available on WWW:
`http://mizar.org/JFM/Vol1/rlvect_1.abs.html`.

[9] Wojciech A. Trybulec, *Lattice of Subgroups of a Group. Frattini Subgroup. Formalized Mathematics* 2(**1**):41–47, 1991. Available on WWW:
`http://mizar.org/JFM/Vol2/group_1.abs.html`.