Automatic Verification of Polynomial Rings Fundamental Properties in ACL2

I. Medina-Bulo[†], J. A. Alonso-Jiménez[*], F. Palomo-Lozano[†]
{inmaculada.medina, francisco.palomo}@uca.es[†]
jalonso@cica.es[*]

Department of Computing Sciences and Artificial Intelligence[*]
University of Sevilla
Department of Computer Languages and Systems[†]
University of Cádiz

Keywords: Computer Algebra, multivariate polynomial, ACL2, NQTHM, automatic reasoning, applicative programming language

Abstract

In this paper we present a formalization of multivariate polynomials over a coefficient field (initially, $\mathbb{Q}$) and of their main properties. This formalization is shown to be adequate for the automatic verification, in an applicative logic like ACL2, of fundamental properties which structure them as a ring, with its main goal being to provide a reusable book on polynomials for the development of further work. As this work comes from a previous formalization attempt in NQTHM, some of the advantages provided by ACL2 regarding this latter system are analyzed in the conclusions.

1 Introduction

Many of the most important algorithms from Computer Algebra [7, 9, 19] work on multivariate polynomials. Several symbolic computation systems have been built over the last fifty years with the aim to automate the growing requirements for the resolution of mathematical problems in Science and Engineering. At the same time, the popularity of these systems has favoured the appearance of new algorithms.

Nevertheless, the kernel of each and every one of these systems develops a solution, usually different, to the problem of representing (efficiently) the different mathematical entities and, particularly, multivariate polynomials and their operations.

In spite of its relevance, it seems that not enough effort has been devoted to computational formalization of the polynomial concept, though there are some works in this direction [2, 14], especially in the context of tools whose reasoning is more guided by the user [3, 18].

This paper tries to cover this lack by providing ACL2 with a reusable book that facilitates structuring further work in automatic verification of polynomial algorithms. Nevertheless, this is not the only goal because it does not only aim at finding a proper formalization for proving, but also for computing (efficiently, as much as possible).

[*]Facultad de Informática y Estadística. Avda. Reina Mercedes, s/n. 41012 Sevilla. Spain.
[†]Escuela Superior de Ingeniería de Cádiz. C/ Chile, s/n. 11003 Cádiz. Spain.

2 Code Organization

Our ACL2 code has been divided into several files that form three packages. A *makefile* is provided with them to automate the process of separate certification. Modular decomposition is shown in table 1.

PACKAGE	FILES	DESCRIPTION
TER	term.lisp	Terms
	lexicographical-ordering.lisp	Lexicographical ordering on terms
MON	monomial.lisp	Monomials
POL	polynomial.lisp	Polynomials
	normal-form.lisp	Polynomial normalization and equality
	addition.lisp	Polynomial addition
	negation.lisp	Polynomial negation
	congruences-1.lisp	Congruences with polynomial construction, addition and negation
	multiplication.lisp	Polynomial multiplication
	congruences-2.lisp	Congruences with polynomial multiplication

Table 1: Logical and physical packaging

These packages import symbols that are typical of ACL2 (those given by `*acl2-exports*` and others), but they do not import arithmetical symbols, nor relational ones, as we have found it more natural to use `+` to stand for addition, `-` for negation, `*` for multiplication, `<` for the ordering relation and `=` for equality. Name resolution mechanism supplied by `defpkg` and `in-package` assures that these names may be used homogeneously in all packages without conflict.

For the sake of brevity, we refrain from including every technical detail of the presented proofs. Instead, we refer the interested reader to the corresponding ACL2 code, available electronically at `www-cs.us.es/~imedina/polynomials.html`.

3 Polynomial Representation

Representation of objects under study is a main concern for the success of any certification work. This is especially important when dealing with polynomials, due to the great variety of representations allowed and to the differences between algorithms that operate them.

The degree of difficulty associated with the automatic proof of a given property depends on the representation chosen to a great extent. Let us describe this in more detail by analyzing the two main representation schemes explored during the development of this work.

3.1 Normalized Representation Problems

Initially we chose a *sparse normalized* representation for polynomials [7, 9, 19], in which a unique representation is associated with each polynomial where neither null coefficient nor identical terms monomials appear.

With this approach, also present in programming of symbolic computation systems, very efficient algorithms may be obtained [10, 19] operating on normalized objects to produce normalized results. To achieve this, it is necessary to define a total strict order on the terms building up monomials, there being various possibilities on the subject that have been widely commented on by many authors.

The main advantage of this method from the verification point of view stems from the fact that semantic equivalence becomes syntactic equality, represented by ACL2's `equal`.

It is not difficult to formalize this representation so that it is admitted by the system. However, problems really appear when trying to prove such elemental properties like associativity of addition. Problems crop up too early with this representation[1].

An exhaustive analysis of failed proofs shows how the most important drawback to this representation is to complicate excessively operation definitions[2] causing a deep impact on proof complexity.

Unfortunately, all this points to the existence of a trade-off between algorithmic efficiency and verification simplicity. This leads us (at the moment) to use another representation, less efficient from the algorithmic point of view, but which makes it easier to verify the properties.

Later, a certain kind of "compositional reasoning" could be used, and we could prove the equivalence of the algorithms used with more efficient versions. In short, the problem is reduced to finding an efficient function for each inefficient function of our representation and to prove that they are equivalent. However, we do not treat this improvement in this work.

3.2 Unnormalized Representation

Using an unnormalized representation presents some drawbacks such as equality is semantic, that is, it has to operate with the equivalence classes induced by the normalization process, and the prover does not manage it directly[3].

Nevertheless, it also has many advantages, because it spares the operations the need of working with normal forms and, therefore, their definitions become greatly simplified. Consequently, the automatic proof of their properties is also easier.

When the computation done by the algorithm is separated from the normalization process, the problem of normalization is concentrated in just one location: the equality predicate. Of course equality becomes complicated, but to a lesser extent than operations with the normalized representation do.

Therefore, the chosen alternative has been one that uses a sparse and unnormalized representation. Note that a dense representation is not appropriate any more, because it does not solve any of the posed problems and it is tremendously space-inefficient (especially when the number of variables is high).

To formalize the problem in ACL2, a polynomial will be represented as a list of monomials and a semantic equality predicate will be defined showing itself as an equivalence relation. Next, main polynomial operations will be defined and we will try to prove the existence of a congruence between each of these operations (for each of its arguments) and the given equivalence relation. Finally, it will be proved that polynomials with these operations have a ring structure.

Each monomial will be represented as a pair (coefficient and term) and a semantic equality predicate will be defined. As we will see later on, terms will be represented by exponent lists, and a multiplication operation and a total ordering relation will be defined on them.

4 Terms

Let $X = \{x_1, \ldots, x_n\}$ be a finite set of variables, with an ordering relation $<_X = \{(x_i, x_j) : 1 \leq i < j \leq n\}$ among its elements.

[1] Concurrent with this work, the authors are developing a framework from which they hope to make feasible this representation through an ACL2's formalization of *typed* finite sets by introducing canonical forms. It is interesting to remark that a recent report by *J S. Moore* on *untyped* finite sets gives some analysis to this matter [13].

[2] Each operation has to deal with the normalization process and, in particular, with keeping the monomials of the resultant polynomial ordered.

[3] This problem may be mitigated in ACL2 by using *congruences*. Other systems, like NQTHM [4, 5] and COQ [8], do not offer this possibility and it is necessary to create *compatibility theorems* between operations and equivalence relations to get something similar (see how this problem influenced on [18]).

Definition. *A term on X is a finite power product of the form:*

$$x_1^{e_1} \ldots x_n^{e_n} = \prod_{i=1}^{n} x_i^{e_i} \qquad \forall i \; e_i \in \mathbb{N},$$

and we will denote it briefly by $X^{\langle e_1,\ldots,e_n\rangle}$.

As we can see later, the main results obtained from this formalization may be summed up in the following points:

1. Terms form a commutative monoid with respect to the multiplication operation.
2. Lexicographical ordering on terms is well-founded and admissible.

A term on X may be represented in an easy way by a list of natural numbers. Its recognizer is given by the following predicate:

```
(defmacro termp (a)
  `(natural-listp ,a))
```

The function `natural-listp`, totally analogous to `integer-listp`, simply checks whether its argument is a true list of elements satisfying the `naturalp` predicate.

```
(defmacro naturalp (x)
  `(and (integerp ,x) (LISP::<= 0 ,x)))

(defun natural-listp (l)
  (cond ((atom l)
         (equal l nil))
        (t
         (and (naturalp (first l))
              (natural-listp (rest l))))))
```

We represent the null term with zero variables by a constant while defining a recognizer for null terms with an arbitrary number of variables.

```
(defconst *null*
  nil)

(defun nullp (a)
  (cond ((atom a)
         (equal a *null*))
        (t
         (and (equal (first a) 0) (nullp (rest a))))))
```

Nevertheless, as we usually work with terms defined on the same set of variables, X, these will be represented by lists of equal length. Therefore, it is appropriate to define a compatibility relation on terms; thus, two terms are said to be *compatible* if they have equal length.

```
(defmacro compatiblep (a b)
  `(equal (len ,a) (len ,b)))
```

4.1 Equality

From the presented definitions it is clear that it is only necessary to define a merely syntactic equality on terms. Nevertheless, for notational purposes, and to make future changes easier, we define the equality symbol[4] as a synonym of `equal`.

```
(defmacro = (a b)
  `(equal ,a ,b))
```

[4] Recall that there is no conflict, because the symbol `=` that we are defining belongs to the package `TER`. This is also applicable to the symbols and packages discussed in section 2.

4.2 Commutative Monoid Structure

Having chosen the set of variables, it suffices to add up their exponents variable by variable to compute the multiplication of two compatible terms.

$$X^{\langle a_1,\ldots,a_n\rangle} \cdot X^{\langle b_1,\ldots,b_n\rangle} = X^{\langle a_1+b_1,\ldots,a_n+b_n\rangle}$$

The following function does this task. Nevertheless, this is somewhat general because the requirement for compatibility inside the function would result in an unnecessary complication.

```
(defun * (a b)
  (declare (xargs :guard (and (termp a) (termp b))))
  (cond ((and (not (termp a)) (not (termp b)))
         *null*)
        ((not (termp a))
         b)
        ((not (termp b))
         a)
        ((endp a)
         b)
        ((endp b)
         a)
        (t
         (cons (LISP::+ (first a) (first b)) (* (rest a) (rest b))))))
```

As shown, elements not being terms behave as if they correspond to the null term. In the case of incompatible terms, the one with less variables is completed; this is the same as assuming that the shorter list is filled with zeros to its right.

A proof of terms having commutative monoid structure with respect to the previous operation is obtained by feeding the system with the following theorems. The generality of the function `*` allows them to be proved while weakening their natural hypothesis.

```
(defthm *-identity-1
  (implies (and (nullp a) (termp b) (compatiblep a b))
           (= (* a b) b)))

(defthm *-identity-2
  (implies (and (termp a) (nullp b) (compatiblep a b))
           (= (* a b) a)))

(defthm commutativity-of-*
  (= (* a b) (* b a)))

(defthm associativity-of-*
  (= (* (* a b) c) (* a (* b c))))
```

Note that it is only necessary to require term compatibility in the two first theorems. For example, if we pay attention to the first one, we see that if `a` were not compatible with `b`, but it had more variables, then the syntactic equality would not follow.

4.3 Well-Ordering

Next, we will show how to define a total and strict order on terms. In addition, this order is proved to be well-founded and admissible.

To order terms, once we have determined the set of variables, X, it is only necessary to take into account exponent lists. The obvious choice is to set up a *lexicographical ordering* among these sequences of natural numbers.

4.3.1 Lexicographical Ordering

In the case of compatible terms, the definition of lexicographical ordering is straightforward, since the natural number sequences involved are of the same length.

$$\langle a_1, \ldots, a_n \rangle < \langle b_1, \ldots, b_n \rangle \equiv \exists i \ (a_i < b_i \wedge \forall j < i \ a_j = b_j)$$

The following boolean function defines the strict lexicographical order relation on terms in this way, but similarly to what happens to `*`, it will be somewhat more general. Thus, if two terms are not compatible, the one with less variables will be taken as the least if, and only if, it is a prefix of the other.

```
(defun < (a b)
  (declare (xargs :guard (and (termp a) (termp b))))
  (cond ((or (endp a) (endp b))
         (not (endp b)))
        ((equal (first a) (first b))
         (< (rest a) (rest b)))
        (t
         (LISP::< (first a) (first b)))))
```

It is not difficult to make evident that the defined relation satisfies the properties of a strict partial ordering (irreflexivity and transitivity).

```
(defthm irreflexivity-of-<
  (not (< a a)))

(defthm transitivity-of-<
  (implies (and (< a b) (< b c)) (< a c)))
```

It is also possible to prove trichotomy, though under somewhat stronger conditions.

```
(defthm trichotomy-of-<
  (implies (and (termp a) (termp b))
           (or (< a b) (< b a) (= a b)))
  :rule-classes nil)
```

However, this property is more useful when stated in the following way, for its corollary can then be used as a rewrite rule.

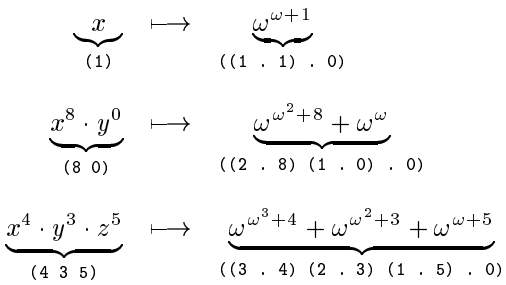
```
(defthm trichotomy-of-<
  (implies (and (termp a) (termp b))
           (or (< a b) (< b a) (= a b)))
  :rule-classes
  ((:rewrite :corollary
             (implies (and (termp a) (termp b)
                           (not (= a b)) (not (< a b)))
                      (< b a)))))
```

4.3.2 Term embedding in ε_0-ordinals

To embed terms in ε_0-ordinals we will adopt the following criterion:

$$X^{\langle e_1, \ldots, e_n \rangle} \longmapsto \omega^{\omega^n + e_1} + \cdots + \omega^{\omega + e_n}$$

This embedding presents the advantage of providing a straightforward translation from the exponents list of the term, as it can be noticed in the examples shown below. On the other hand, the obtained ordinal type makes this representation easy to handle.



We proceed to embed terms in ε_0-ordinals by using the following function.

```
(defun term->e0-ordinal (a)
  (declare (xargs :guard (termp a)))
  (cond ((endp a)
         0)
        (t
         (cons (cons (len a) (first a))
               (term->e0-ordinal (rest a))))))
```

As we will see next, it is proved that `term->e0-ordinal` truly produces an ε_0-ordinal from a term.

4.3.3 Well-Foundedness

To state that a relation is well-founded in ACL2, it is first necessary to make available a function to perform the embedding of the relation objects in ε_0-ordinals. However, it is very important to prove the correctness of the embedding function, which is not always easy when its ordinal type is high. In this case, it is not a hard task after proving a technical lemma:

```
(encapsulate ()
  (local
    (defthm technical-lemma
      (implies (and (termp a)
                    (e0-ordinalp (term->e0-ordinal (rest a))))
               (e0-ordinalp (term->e0-ordinal a)))
      :otf-flg t))

    (defthm e0-ordinalp-term->e0-ordinal
      (implies (termp a)
               (e0-ordinalp (term->e0-ordinal a)))
      :hints (("Goal"
               :in-theory (disable e0-ordinalp term->e0-ordinal)))))
```

Once the correction of the embedding function has been proved, it is enough to check that it preserves the order, that is to say, that the ε_0-ordinals corresponding to each pair of related elements remain related.

```
(defthm well-ordering-of-<
  (and (implies (termp a)
                (e0-ordinalp (term->e0-ordinal a)))
       (implies (and (termp a) (termp b)
                     (< a b))
                (e0-ord-< (term->e0-ordinal a) (term->e0-ordinal b))))
  :rule-classes :well-founded-relation)
```

This procedure allows us in ACL2 to add the rule class `:well-founded-relation` to the well-foundedness theorem, thus marking the defined ordering relation (which is *Noetherian*) to be used, when necessary, to prove the strict decrease of a measure function in the domain of terms.

Unfortunately, when using the presented < function, this theorem cannot be proved, for it is false. In fact, it suffices to consider terms with a different number of variables to understand the problem; there are clearly two symmetric cases, depending on whether the first has less variables than the second or vice versa:

$$\begin{array}{ccc} x^4y^2z & <_X & x^6y^4 \\ \downarrow & & \downarrow \\ \omega^{\omega^3+4}+\omega^{\omega^2+2}+\omega^{\omega+1} & \not<_{\varepsilon_0} & \omega^{\omega^2+6}+\omega^{\omega+4} \end{array} \qquad \begin{array}{ccc} x^8 & \not<_X & x^3y^2 \\ \downarrow & & \downarrow \\ \omega^{\omega+8} & <_{\varepsilon_0} & \omega^{\omega^2+3}+\omega^{\omega+2} \end{array}$$

When terms are compatible, the problem disappears. It could be thought that the adequate completion of the term with less variables could avoid the problem. However, the solution is not as simple, because when embedding a term nothing is known about which other terms it could be compared to. A feasible solution is to deal especially with both cases:

```
(defun < (a b)
  (declare (xargs :guard (and (termp a) (termp b))))
  (cond ((LISP::< (len a) (len b))
         t)
        ((LISP::> (len a) (len b))
         nil)
        ( ...
```

Now, we can prove that this relation is well-founded.

4.3.4 Admissibility

Finally, it is stated that the order is admissible on the set of compatible terms. For that, the existence of a first element is proved (in fact, it is proved that every null term acts as the first element) and that it is compatible with the operations, in this case, just the multiplication.

```
(defthm <-has-first
  (implies (and (termp a) (termp b)
                (compatiblep a b)
                (nullp a) (not (nullp b)))
           (< a b)))

(defthm <-compatible-*-1
  (implies (and (termp a) (termp b) (termp c)
                (compatiblep a c) (compatiblep b c)
                (< a b))
           (< (* a c) (* b c))))

(defthm <-compatible-*-2
  (implies (and (termp a) (termp b) (termp c)
                (compatiblep a c) (compatiblep b c)
                (< a b))
           (< (* c a) (* c b))))
```

To demand term compatibility is essential, due to the change made in the original definition of the < function.

5 Monomials

Definition. *A monomial on X is a product of the form $c \cdot X^{\langle e_1,\ldots,e_n\rangle}$, where c is called the coefficient and e_i are called the exponents. The $\cdot$ operation is defined from the set of coefficients to the set of values that the elements in X can take.*

Note that, for our purposes, it is not necessary to define the set on which the elements in X take their values; these elements may be regarded as formal symbols with an indeterminate meaning. We will use the field $\mathbb{Q}$ for the coefficients, although other algebraic systems could have been used[5].

Clearly, to represent a monomial it suffices to use a list whose first element is its coefficient and whose rest is the accompanying term.

A very simple formalization in ACL2 is got by using macros, because, in fact, the concept of monomial merely exists for notational easiness. The constructor and accessor operations are defined in the following way:

```
(defmacro monomial (c e)
  `(cons ,c ,e))

(defmacro coefficient (a)
  `(first ,a))

(defmacro term (a)
  `(rest ,a))
```

It is also necessary to define a recognizer that allows us to discern which ACL2 objects are monomials and which are not:

```
(defmacro monomialp (a)
  `(and (consp ,a)
        (rationalp (first ,a))
        (termp (rest ,a))))
```

Multiplicative identity monomial with null term is defined by a constant. To define a recognizer for multiplicative identity monomials it suffices to create a macro that checks if the coefficient is 1 and the accompanying term is null.

```
(defconst *one*
  (monomial 1 TER::*null*))

(defmacro onep (a)
  `(and (equal (coefficient ,a) 1)
        (TER::nullp (term ,a))))
```

Also, it is handy to define a constant to represent the null monomial and a recognizer for null monomials. Any monomial whose coefficient is null will be recognized as such.

```
(defconst *null*
  (monomial 0 TER::*null*))

(defmacro nullp (a)
  `(equal (coefficient ,a) 0))
```

In the same way as with terms, it is suitable to define a compatibility relation on monomials. Two monomials are compatible if, and only if, their underlying terms are compatible. It is obvious that the relation defined in this way is an equivalence.

```
(defun compatiblep (a b)
  (declare (xargs :guard (and (monomialp a) (monomialp b))))
  (TER::compatiblep (term a) (term b)))

(defequiv compatiblep)
```

[5] The book may be certified without any problem after replacing `rationalp` with `integerp`, thus obtaining integer coefficient polynomials, or with `acl2-numberp`, in which case the coefficients become complex rationals.

5.1 Monoid commutative structure

To compute the multiplication of two monomials it suffices to multiply their coefficients and their terms.

```
(defun * (a b)
  (declare (xargs :guard (and (monomialp a) (monomialp b))))
  (monomial (LISP::* (coefficient a) (coefficient b))
            (TER::* (term a) (term b))))
```

Monomials inherit trivially a commutative monoid structure from terms and from properties of the coefficient field multiplication operation.

```
(defthm *-identity-1
  (implies (and (onep a) (monomialp b) (compatiblep a b))
           (= (* a b) b)))

(defthm *-identity-2
  (implies (and (monomialp a) (onep b) (compatiblep a b))
           (= (* a b) a)))

(defthm associativity-of-*
  (= (* (* a b) c) (* a (* b c)))
  :hints (("Goal"
           :in-theory (disable ACL2::commutativity-of-*))))

(defthm commutativity-of-*
  (= (* a b) (* b a)))
```

The cancellation properties of monomial multiplication are proved without any difficulty.

```
(defthm *-cancellative-1
  (implies (and (nullp a) (compatiblep a b))
           (nullp (* a b))))

(defthm *-cancellative-2
  (implies (and (nullp b) (compatiblep a b))
           (nullp (* a b))))
```

5.2 Semantic Equality and Congruence

Two monomials are equal if they are both null, or if their coefficients and terms are respectively equal. This relation is proved to be an equivalence and a congruence with the multiplication operation in both arguments.

```
(defun = (a b)
  (declare (xargs :guard (and (monomialp a) (monomialp b))))
  (or (and (nullp a) (nullp b))
      (and (LISP::= (coefficient a) (coefficient b))
           (TER::= (term a) (term b)))))

(defequiv =)

(defcong = = (* a b) 1)
(defcong = = (* a b) 2)
```

6 Polynomials

Definition. *A polynomial on X is a finite sum of monomials:*

$$c_1 \cdot X^{\langle e_{11},\ldots,e_{1n}\rangle} + \cdots + c_m \cdot X^{\langle e_{m1},\ldots,e_{mn}\rangle} = \sum_{i=1}^{m} c_i \cdot X^{\langle e_{i1},\ldots,e_{in}\rangle}$$

We begin by defining the recognizer for polynomials. A polynomial is simply represented by a list of monomials.

$$\langle (c_1, \langle e_{11}, \ldots, e_{1n}\rangle), \ldots, (c_m, \langle e_{m1}, \ldots, e_{mn}\rangle)\rangle$$

```
(defun monomial-listp (l)
  (cond ((atom l)
         (equal l nil))
        (t
         (and (monomialp (first l))
              (monomial-listp (rest l))))))

(defmacro polynomialp (p)
  `(monomial-listp ,p))
```

The null polynomial with no monomials is defined as a constant and it is recognized by a macro that is adequate for its use in base cases of recursion.

```
(defconst *null*
  nil)

(defmacro nullp (p)
  `(endp ,p))
```

The constructor simply adds a monomial to a polynomial, although this is defined in such a way that anomalous cases are dealt with in a reasonable way. This is essential to enable a later definition of congruences with it.

```
(defun polynomial (m p)
  (declare (xargs :guard (and (monomialp m) (polynomialp p))))
  (cond ((and (not (monomialp m)) (not (polynomialp p)))
         *null*)
        ((not (monomialp m))
         p)
        ((not (polynomialp p))
         (list m))
        (t
         (cons m p))))
```

Compatibility of monomials must be extended to polynomials. To achieve this we begin by defining the concept of *uniform polynomial*. A polynomial is said to be uniform if all of its monomials are compatible with each other.

```
(defun uniformp (p)
  (declare (xargs :guard (polynomialp p)))
  (or (nullp p)
      (nullp (rest p))
      (and (MON::compatiblep (first p) (first (rest p)))
           (uniformp (rest p)))))
```

Another related concept is that of a *complete polynomial*. A polynomial is complete with n variables, if all of its monomials have got terms with n variables.

```
(defun completep (p n)
  (declare (xargs :guard (and (polynomialp p) (naturalp n))))
  (or (nullp p)
      (and (equal (len (term (first p))) n)
           (completep (rest p) n))))
```

As a consequence of these definitions we conclude that a polynomial is uniform if, and only if, it is complete.

```
(defthm uniformp-iff-completep
  (iff (uniformp p) (completep p (len (term (first p)))))
  :rule-classes nil)
```

Thus the definition of compatibility between polynomials now arises in a natural way. Two polynomials are *compatible* if they are uniform and their two first monomials are compatible too.

```
(defmacro compatiblep (p1 p2)
  `(and (uniformp ,p1) (uniformp ,p2)
        (MON::compatiblep (first ,p1) (first ,p2))))
```

Let us remark that the operations we will define on polynomials will be generalized to properly handle any ACL2 object, even non-polynomials. A non-polynomial object will be regarded as being a null polynomial, rendering the logic on polynomials total. Thanks to this, it is possible to state most of congruence theorems with operations, because `defcong` does not allow for any restrictive hypothesis over the involved objects.

It must not be forgotten that this in no way prevents the specification of guards adequate to the character of each function, because these lack logical significance. Thus, executable versions of functions may be more efficient, for they are allowed to assume that they receive polynomial objects[6].

6.1 Semantic Equality

To decide whether two polynomials are semantically equivalent, we must check that both belong to the same class of equivalence. This is done by computing their normal forms (that is, the canonical representatives of their respective equivalence classes) and examining whether they are syntactically equal. A uniform polynomial is said to be in normal form if it satisfies the following conditions:

1. Its monomials are strictly ordered by a decreasing term order.
2. It contains no null monomial.

Note that the first condition implies the non-existence of monomials with identical terms in a normalized uniform polynomial.

Initially, a function capable of adding a monomial to a polynomial is defined. This function will be such that, if the polynomial is a normalized one, its result will also be a normalized one. For this function to be total we need to complete, taking the utmost care, the values it must return outside the domain set by its guard.

```
(defun +-monomial (m p)
  (declare (xargs :guard (and (monomialp m) (polynomialp p))))
  (cond ((and (not (monomialp m)) (not (polynomialp p)))
         *null*)
        ((not (monomialp m))
         p)
```

[6] In general, if an operation is executed outside its domain in a COMMON LISP system without run-time guard-checking, its behavior is, in the best case, system dependent.

```
((and (not (polynomialp p)) (MON::nullp m))
 *null*)
((not (polynomialp p))
 (polynomial m *null*))
((MON::nullp m)
 p)
((nullp p)
 (polynomial m *null*))
((TER::= (term m) (term (first p)))
 (let ((c (LISP::+ (coefficient m) (coefficient (first p)))))
   (if (equal c 0)
       (rest p)
     (polynomial (monomial c (term m)) (rest p)))))
((TER::< (term (first p)) (term m))
 (polynomial m p))
(t
 (polynomial (first p) (+-monomial m (rest p))))))
```

From this function, the computation of normal forms can be defined. If the polynomial is null, it is already in normal form; therefore it suffices to normalize the rest of the polynomial if it is not a null one and to add its first monomial to the result by using the previous function.

```
(defun nf (p)
  (declare (xargs :guard (polynomialp p)))
  (cond ((or (not (polynomialp p)) (nullp p))
         *null*)
        (t
         (+-monomial (first p) (nf (rest p))))))
```

Having done this, it is easy to prove that the equality relation defined on polynomials is an equivalence.

```
(defun = (p1 p2)
  (declare (xargs :guard (and (polynomialp p1) (polynomialp p2))))
  (equal (nf p1) (nf p2)))

(defequiv =)
```

Other important properties have been proved, such as that the normalization function developed meets its specification and that uniformity and completeness of a polynomial are preserved after transforming it into a normal form. The reader is referred to the corresponding code.

6.2 Commutative Ring Structure

Next, operations allowing us to add, multiply and negate polynomials will be defined. To ensure that these operations satisfy the fundamental properties everybody expects from them, with the representation chosen for polynomials, the existence of a commutative ring structure must be proved.

Therefore, it is necessary to check that polynomials with addition and negation form an Abelian group, while forming a commutative monoid with multiplication; besides that, multiplication must distribute over addition.

6.2.1 Commutative Group with Addition and Negation

To add two polynomials it suffices to append their monomial lists. In fact, this is the easiest way of defining this operation and it presents the advantage of simplifying the associative property proof a lot. If getting the reduced result is what is desired, it is sufficient to compute its normal form.

```
(defun + (p1 p2)
  (declare (xargs :guard (and (polynomialp p1) (polynomialp p2))))
  (cond ((and (not (polynomialp p1)) (not (polynomialp p2)))
         *null*)
        ((not (polynomialp p1))
         p2)
        ((not (polynomialp p2))
         p1)
        (t
         (append p1 p2))))
```

To compute the negative of a polynomial you only need to replace the coefficient in each monomial with its negative.

```
(defun - (p)
  (cond ((or (not (polynomialp p)) (nullp p))
         *null*)
        (t
         (polynomial (monomial (LISP::- (coefficient (first p)))
                               (term (first p)))
                     (- (rest p))))))
```

It is not hard to prove that this operation distributes over the addition of polynomials.

```
(defthm --distributes-+
  (= (- (+ p1 p2)) (+ (- p1) (- p2))))
```

The following theorems prove that polynomials with the aforesaid operations have a group structure.

```
(defthm +-identity-1
  (= (+ p *null*) p))

(defthm +-identity-2
  (= (+ *null* p) p))

(defthm associativity-of-+
  (= (+ (+ p1 p2) p3) (+ p1 (+ p2 p3))))

(defthm +--
  (= (+ p (- p)) *null*)))
```

It is much more complex to prove group commutativity than the other properties.

```
(defthm commutativity-of-+
  (= (+ p1 p2) (+ p2 p1))
  :hints (("Goal"
           :in-theory (disable =))))
```

6.2.2 Commutative Monoid with Multiplication

The multiplicative identity polynomial in normal form is defined as a constant. Elements in its equivalence class can be recognized by a simple macro.

```
(defconst *one*
  (polynomial MON::*one* *null*))

(defmacro onep (p)
  `(= ,p *one*))
```

Before defining the internal multiplication operation between polynomials it is feasible to define a helper function to represent the external multiplication between monomials and polynomials. The procedure consists of replacing each monomial from the original polynomial with its multiplication by the given monomial.

```
(defun *-monomial (m p)
  (declare (xargs :guard (and (monomialp m) (polynomialp p))))
  (cond ((or (nullp p) (not (monomialp m)) (not (polynomialp p)))
         *null*)
        (t
         (polynomial (MON::* m (first p)) (*-monomial m (rest p))))))
```

It is proved that this auxiliary operation has an identity and a cancellative element and that it is distributive over the addition of monomial and polynomial, as well as over polynomial addition. Then, to compute the multiplication of two polynomials it is enough to use the following definition:

```
(defun * (p1 p2)
  (declare (xargs :guard (and (polynomialp p1) (polynomialp p2))))
  (cond ((or (nullp p1) (not (polynomialp p1)))
         *null*)
        (t
         (+ (*-monomial (first p1) p2) (* (rest p1) p2)))))
```

The fact that polynomials with this multiplication operation have monoid structure is deduced from the following theorems.

```
(defthm *-identity-1
  (= (* *one* p) p))

(defthm *-identity-2
  (= (* p *one*) p))

(defthm associativity-of-*
  (= (* p1 (* p2 p3)) (* (* p1 p2) p3))
  :hints (("Goal"
           :in-theory (disable = +))))
```

It is more complex to prove monoid commutativity. Its proof is an example of the usefulness of congruences defined between equality and the addition operation. The proof requires the definition of a suitable induction scheme, a previously proved technical lemma, and several properties.

```
(defthm commutativity-of-*
  (= (* p1 p2) (* p2 p1))
  :hints (("Goal"
           :induct (induction-scheme p1 p2)
           :do-not '(eliminate-destructors)
           :in-theory (disable = + polynomial))))
```

Properties stating the existence of cancellative elements in both arguments do not present any difficulty for the prover.

```
(defthm *-cancellative-1
  (= (* *null* p) *null*))

(defthm *-cancellative-2
  (= (* p *null*) *null*))
```

6.2.3 Distributivity of Multiplication over Addition

Finally, we are capable to prove that polynomial multiplication is distributive over addition.

```
(defthm *-distributes-+-1
  (= (* p1 (+ p2 p3)) (+ (* p1 p2) (* p1 p3)))
  :hints (("Goal"
           :in-theory (disable = +))))

(defthm *-distributes-+-2
  (= (* (+ p1 p2) p3) (+ (* p1 p3) (* p2 p3)))
  :hints (("Goal"
           :in-theory (disable nf + *)))))
```

6.3 Congruences

One of the most interesting aspects of the formalization chosen here is that it allows the definition, in most cases, of congruences between the equivalence relation given by polynomial equality under normal form and their operations. This feature noticeably increases chances of reusing the book as a tool for proving higher level properties.

The first operation, with which congruences can be set up, is the constructor of polynomial objects. In this case, two equivalence relations intervene: those defined on monomials and on polynomials. Both congruences are stated without difficulties.

```
(defcong MON::= = (polynomial m p) 1)
(defcong = = (polynomial m p) 2)
```

A bit more complex is the congruence with negation, where a technical lemma is necessary to allow the normalization process to be "pushed" into the operation.

```
(encapsulate ()
  (local
    (defthm technical-lemma
      (equal (nf (- p)) (- (nf p)))))

  (defcong = = (- p) 1))
```

The technique used is similar in the case of the addition, with the exception that the normalization process cannot be eliminated not even if it is also introduced into the argument. This leads to the requirement of some little hints to carry out the proofs.

```
(encapsulate ()
  (local
    (defthm technical-lemma-1
      (= (+ p1 (nf p2)) (+ p1 p2))))

  (defcong = = (+ p1 p2) 2
    :hints (("Goal"
             :in-theory (disable technical-lemma-1)
             :use ((:instance technical-lemma-1 (p2 ACL2::p2-equiv))
                   technical-lemma-1))))

  (local
    (defthm technical-lemma-2
      (= (+ (nf p1) p2) (+ p1 p2))
      :hints (("Goal"
               :in-theory (disable =)))))
```

```
(defcong = = (+ p1 p2) 1
  :hints (("Goal"
           :in-theory (disable technical-lemma-2)
           :use ((:instance technical-lemma-2 (p1 ACL2::p1-equiv))
                 technical-lemma-2)))))
```

Proving that the multiplication of monomial and polynomial is congruent with respect to the equality in the first argument is direct.

```
(defcong MON::= = (*-monomial m p) 1)
```

However, problems with .defcong. crop up while trying to create congruences in the second argument of this operation. With the foregoing definitions it is necessary to keep the compatibility hypothesis, and this prevents `:congruence` from being accepted as a legal rule class, for it does not have the proper form. Moreover, the proof is noticeably more elaborate.

```
(defthm =-implies-=-*-monomial-2
  (implies (and (monomialp m) (polynomialp p1) (polynomialp p1-equiv)
                (MON::compatiblep m (first p1)) (compatiblep p1 p1-equiv)
                (= p1 p1-equiv))
           (= (*-monomial m p1) (*-monomial m p1-equiv)))
  :hints (("Goal"
           :cases ((MON::nullp m) (not (MON::nullp m))))
          ("Subgoal 2"
           :in-theory (disable =)))))
```

Then, this defect is extended to the polynomial multiplication operation, because this latter is derived from it. Proofs are still more intricate, and need several technical lemmas along with some hints to be performed.

```
(defthm =-implies-=-*-2
  (implies (and (polynomialp p1) (polynomialp p2) (polynomialp p2-equiv)
                (compatiblep p1 p2) (compatiblep p1 p2-equiv)
                (= p2 p2-equiv))
           (= (* p1 p2) (* p1 p2-equiv)))
  :hints (("Goal"
           :in-theory (disable nf-*-1)
           :use (nf-*-1
                 (:instance nf-*-1 (p2 p2-equiv))))))

(defthm =-implies-=-*-1
  (implies (and (polynomialp p1) (polynomialp p1-equiv) (polynomialp p2)
                (compatiblep p1 p2) (compatiblep p1-equiv p2)
                (= p1 p1-equiv))
           (= (* p1 p2) (* p1-equiv p2)))
  :hints (("Goal"
           :in-theory (disable commutativity-of-*)
           :use (commutativity-of-*
                 (:instance commutativity-of-* (p1 p1-equiv))
                 (:instance =-implies-=-*-2 (p1 p2) (p2 p1) (p2-equiv p1-equiv))))))
```

6.4 Examples

As an example of some of the elemental properties that can be automatically proved by using the book on polynomials developed here, without the need of providing the prover with any hint, we present the following, that state how the polynomial semantic equality is preserved under several conditions.

```
(defthm polynomial-first-null
  (implies (MON::nullp m)
           (= (polynomial m p) p)))

(defthm +-polynomial
  (= (+ p1 (polynomial m p2)) (polynomial m (+ p1 p2))))

(defthm polynomial-polynomial-monomial-=-term
  (implies (and (monomialp m1) (monomialp m2) (polynomialp p)
                (TER::= (term m1) (term m2)))
           (= (polynomial m1 (polynomial m2 p))
              (polynomial (monomial (LISP::+ (coefficient m1) (coefficient m2))
                                    (term m1))
                          p))))
```

7 Conclusions and Future Work

A formalization of multivariate polynomials rings with rational coefficients in ACL2 has been presented. This includes a lexicographical ordering on terms along with its proofs of admissibility and well-foundedness, besides a normalization function and an induced equivalence relation on which congruences suitable for attacking harder problems are stated.

It must be noted that some of the obtained theorems have rather complex proofs helped by technical lemmas and additional and by no means easy properties. Table 2 shows, for the sake of comparison, the number of lines of code in each file, those generated by the system during its certification and the proportion of time measured[7] with respect to the one in which execution takes less.

FILE	LINES OF CODE	LINES OF PROOF	PROP. OF TIME
term.lisp	147	2019	9.8
lexicographical-ordering.lisp	135	4996	43.0
monomial.lisp	151	738	1.5
polynomial.lisp	115	852	1.0
normal-form.lisp	164	12970	169.0
addition.lisp	94	1467	15.7
negation.lisp	52	1454	15.5
congruences-1.lisp	55	1160	28.3
multiplication.lisp	213	12619	268.8
congruences-2.lisp	170	3337	106.0

Table 2: Certification statistics

This work comes from an unfinished previous one [12], in which the *Boyer-Moore* theorem prover, NQTHM, was used. It is interesting to note some of the advantages exposed by ACL2 in comparison with NQTHM that influenced our decision of translating the problem into the former (in spite of certain initial reticences due to the disappearance of *shells*, for we used them profusely).

The main inconvenience that cropped up when formalizing polynomials stemmed from the chosen coefficient field. NQTHM contains only natural numbers, so we had to formalize and implement a coefficient field from scratch.

In short, since ACL2 already incorporates a proper formalization of $\mathbb{Q}$, the prerequisite to our efforts for yielding a suitable representation of polynomials has been satisfied.

[7] Version 2.4 of ACL2 was used. Times are measured and divided by the smallest one to get a ratio more or less independent of the system in which the certification is done.

Problems derived from the absence of *shells* were solved by a more careful formalization and proper choice of type prescription rules.

On the other hand, NQTHM's absence of congruences implied the continuous need for proving trivial lemmas, which, at first sight, had little or no relation to the theorems we really wanted to prove. In the subsequent development under ACL2, congruences played their role, either shortening the length and time of several proofs, or eliminating unnecessary hints which reduced their degree of automation. The commutativity of polynomial multiplication proof is a true example of this.

Unfortunately, the introduction of the concept of term compatibility, arising from the need of proving the well-foundedness of lexicographical ordering on terms, prevented us from entirely achieving this goal by avoiding the elimination of compatibility hypothesis in the case of multiplication.

A possible solution is to define the operations so that they work the usual way only on compatible polynomials, while assigning an arbitrary meaning in cases involving incompatible polynomials (in analogy to what now happens when dealing with non-polynomial objects). Another possible avenue would be to find a suitable formalization of terms and of their embedding in ε_0-ordinals that would totally eliminate the need to bring up the concept of term compatibility.

It is worthwhile to note ACL2's support for guards to indicate function preconditions and, especially, its automatic verification that guarantees that the COMMON LISP code can be executed efficiently, and with the same results, on any platform.

Last but not least, we would like to remark that this work is related to others developed at Sevilla University Computational Logic Group on mechanized proving of several algorithms and theorems from Rewrite Theory (see [1] for a precise and modern description of this theory). Particularly, algorithms for subsumption, unification and anti-unification have been certified [15, 16], and *Knuth-Bendix* critical pair theorem has been mechanically proved [17].

As a matter of fact, this is only part of a rather more ambitious ongoing project, whose aim is to obtain an automatic verification of *Buchberger*'s algorithm for *Gröbner* bases computation in ACL2. Works from [3, 18], certainly complementary, achieve this goal in COQ. Nevertheless, ACL2 and COQ logics differ in many aspects and automation degree achievable in ACL2 is, at first sight, superior to COQ.

The reduction relation on polynomials defined for *Buchberger*'s algorithm is a subset of the ordering on polynomials induced by lexicographical ordering stated on terms. Consequently, defining a term order is only the first step in defining the concepts associated with *Buchberger*'s algorithm and, particularly, to its proof of termination.

There are many applications of *Gröbner* bases, but we are mainly concerned with one that is directly related to Propositional Logic. In Classical Propositional Logic, a polynomial is associated with each formula by *Stone* isomorphism. Then, tautology and deduction problems can be solved after computing a given *Gröbner* basis. This "algebraic method" has been extended to finite Multi-Valued Propositional Logic [6, 20].

References

[1] *Baader, F. & Nipkow, T. Term Rewriting and All That.* Cambridge University Press. 1998.

[2] *Ballarin, C. Computer Algebra and Theorem Proving.* Technical Report, 473. University of Cambridge Computer Laboratory. 1999.
`iaks-www.ira.uka.de/iaks-calmet/ballarin`

[3] *Barja-Pérez, J. M. & Pérez-Vega, G. Demostración en Implementaciones Concretas de Anillos de Polinomios.* RSMAE. 1999.

[4] *Boyer, R. S. & Moore, J S. A Computational Logic.* Academic Press. 1978.

[5] *Boyer, R. S. & Moore, J S. A Computational Logic Handbook.* Academic Press. 2nd ed. 1998.

[6] *Chazarain, J.; Riscos, A.; Alonso, J. A. & Briales, E.. Multi-Valued Logic and Gröbner Bases with Applications to Modal Logic.* Journal of Symbolic Computation. Vol. 11. 1991.

[7] *Davenport, J. H.; Siret, Y. & Tournier, E. Computer Algebra. Systems and Algorithms for Algebraic Computation.* Academic Press. 1988.

[8] *Dowek, G; Felty, A.; Herbelin, H.; Huet, G.; Murty, C; Parent, C; Paulin-Mohring, C. & Werner, B. The* Coq *Proof Assistant Reference Manual.* Rapport Technique, 0203. INRIA. 1999.

[9] *Geddes, K. O.; Czapor, S. R. & Labahn, G. Algorithms for Computer Algebra.* Kluwer. 1992.

[10] *Johnson, S. C. Sparse Polynomial Arithmetic.* SIGSAM Bulletin 8. 1974.

[11] *Kaufmann, M. & Moore, J S. An Industrial Strength Theorem Prover for a Logic Based on* Common Lisp. IEEE Transactions on Software Engineering. 1997.

[12] *Medina-Bulo, I. Verificación de Propiedades de Algoritmos Polinómicos.* Informe Técnico. CCIA. Universidad de Sevilla. 2000.

[13] *Moore, J S. Finite Set Theory in* Acl2. Technical Report. Department of Computer Sciences. University of Texas.
`www.cs.utexas.edu/users/moore/publications/finite-set-theory`

[14] *Persson, H. Certified Computer Algebra.* Lecture Notes. Types Summer School 99. 1999.

[15] *Ruiz-Reina, J. L.; Alonso-Jiménez, J. A.; Hidalgo-Doblado, M. J. & Martín-Mateos F. J. Mechanical Verification of a Rule-Based Unification Algorithm in the Boyer-Moore Theorem Prover.* AGP'99 Joint Conference on Declarative Programming. 1999.

[16] *Ruiz-Reina, J. L.; Alonso-Jiménez, J. A.; Hidalgo-Doblado, M. J. & Martín-Mateos F. J. Mechanical Verification of Knuth-Bendix Critical Pair Theorem (using ACL2).* FTP 2000 Third International Workshop on First-Order Theorem Proving. Research Report 5/2000, Universitat Koblenz-Landau.

[17] *Ruiz-Reina, J. L.; Alonso-Jiménez, J. A.; Hidalgo-Doblado, M. J. & Martín-Mateos F. J. Formalizing Rewriting in the ACL2 Theorem Prover.* Fifth International Conference on Artificial Intelligence and Symbolic Computation, 2000. LNCS, Springer-Verlag (to appear).

[18] *Théry, L. A Certified Version of Buchberger's Algorithm.* LNAI, 1421. Springer-Verlag. 1998.

[19] *Winkler, F. Polynomial Algorithms in Computer Algebra.* Springer-Verlag. 1996.

[20] *Wu, J. First-Order Polynomial based Theorem Proving.* In *Mathematics Mechanization and Applications.* (*X-S. Gao* and *D. Wang*, eds.) Academic Press. 1999.