Verified Computer Algebra in ACL2
(Gröbner Bases Computation)

I. Medina-Bulo[1], F. Palomo-Lozano[1],
J.A. Alonso-Jiménez[2], and J.L. Ruiz-Reina[2]

[1] Depto. de Lenguajes y Sistemas Informáticos, Univ. de Cádiz
E.S. de Ingeniería de Cádiz, C/ Chile, s/n, 11003 Cádiz, España
{inmaculada.medina,francisco.palomo}@uca.es
[2] Depto. de Ciencias de la Computación e Inteligencia Artificial, Univ. de Sevilla
E.T.S. de Ingeniería Informática, Avda. Reina Mercedes, s/n, 41012 Sevilla, España
{jalonso,jruiz}@us.es

Abstract. In this paper, we present the formal verification of a Common Lisp implementation of Buchberger's algorithm for computing Gröbner bases of polynomial ideals. This work is carried out in the ACL2 system and shows how verified Computer Algebra can be achieved in an executable logic.

1 Introduction

Computer Algebra has experienced a great development in the last decade, as can be seen from the proliferation of Computer Algebra Systems (CAS). These systems are the culmination of theoretical results obtained in the last half century. One of the main achievements is due to B. Buchberger. In 1965 he devised an algorithm for computing Gröbner bases of multivariate polynomial ideals, thus solving the ideal membership problem for polynomial rings. Currently, his algorithm is available in most CAS and its theory, implementation and numerous applications are widely documented in the literature, e.g. [2, 4].

The aim of this paper is to describe the formal verification of a naive Common Lisp implementation of Buchberger's algorithm. The implementation and formal proofs have been carried out in the ACL2 system, which consists of a programming language, a logic for stating and proving properties of the programs, and a theorem prover supporting mechanized reasoning in the logic.

The importance of Buchberger's algorithm in Computer Algebra justifies on its own the effort of obtaining a formal correctness proof with a theorem prover, and this is one of the motivations for this work. Nevertheless, this goal has already been achieved by L. Théry in [13], where he gives a formal proof using the Coq system and explains how an executable implementation in the Ocaml language is extracted from the algorithm defined in Coq. In contrast, in ACL2 we can reason directly about the Lisp program implementing the algorithm, i.e. about the very program which is executed by the underlying Lisp system. There is a price to pay: the logic of ACL2 is a quantifier-free fragment of first-order

logic, less expressive[1] than the logic of Coq, which is based on type theory. We show how it is possible to formalize all the needed theory within the Acl2 logic.

The formal proofs developed in Acl2 are mainly adapted from Chap. 8 of [1]. As the whole development consists of roughly one thousand Acl2 theorems and function definitions, we will only scratch its surface presenting the main results and a sketch of how the pieces fit together. We will necessarily omit many details that, we expect, can be inferred from the context.

2 The Acl2 System

Acl2 formalizes an applicative subset of Common Lisp. In fact, the same language, based on prefix notation, is used for writing Lisp code and stating theorems about it[2]. The logic is a quantifier-free fragment of first-order logic with equality. It includes axioms for propositional logic and for a number of Lisp functions and data types. Inference rules include those for propositional calculus, equality and instantiation (variables in formulas are implicitly universally quantified). One important inference rule is the *principle of induction*, that permits proofs by well-founded induction on the ordinal ϵ_0 (the logic provides a constructive definition of the ordinals up to ϵ_0).

By the *principle of definition* new function definitions are admitted as axioms (using `defun`) only if its termination is proved by means of an ordinal measure in which the arguments of each recursive call, if any, decrease. In addition, the *encapsulation principle* allows the user to introduce new function symbols (using `encapsulate`) that are constrained to satisfy certain assumptions. To ensure that the constraints are satisfiable, the user must provide a witness function with the required properties. Within the scope of an `encapsulate`, properties stated as theorems need to be proved for the witnesses; outside, these theorems work as assumed axioms. Together, encapsulation and the derived inference rule, *functional instantiation*, provide a second-order aspect [5, 6]: theorems about constrained functions can be instantiated with function symbols if they are proved to have the same properties.

The Acl2 theorem prover mechanizes the logic, being particularly well suited for obtaining automated proofs based on simplification and induction. Although the prover is automatic in the sense that once a proof attempt is started (with `defthm`) the user can no longer interact, nevertheless it is interactive in a deeper sense: usually, the role of the user is to lead the prover to a preconceived hand-proof, by proving a suitable collection of lemmas that are used as rewrite rules in subsequent proofs (these lemmas are usually discovered by the user after the inspection of failed proofs). We used this kind of interaction to obtain the formal proofs presented here. For a detailed description of Acl2, we refer the reader to the Acl2 book [5].

[1] Nevertheless, the degree of automation of the Acl2 theorem prover is higher than in other systems with more expressive logics.

[2] Although we are aware that prefix notation may be inconvenient for people not used to Lisp, we will maintain it to emphasize the use of a *real* programming language.

3 Polynomial Rings and Ideals

Let $R = K[x_1, \ldots, x_k]$ be a polynomial ring on an arbitrary commutative field K, where $k \in \mathbb{N}$. The elements of R are polynomials in the indeterminates $x_1, \ldots, x_k$ with the coefficients in K. Polynomials are built from monomials of R, that is, power products like $c \cdot x_1^{a_1} \cdots x_k^{a_k}$, where $c \in K$ is the coefficient, $x_1^{a_1} \cdots x_k^{a_k}$ is the term, and $a_1, \ldots, a_k \in \mathbb{N}$.

Therefore, there are several algebraic structures that it is necessary to formalize prior to the notion of polynomial. A computational theory of multivariate polynomials on a coefficient field was developed in [8, 9]. This ACL2 formalization includes common operations and fundamental properties establishing a ring structure. The aim was to develop a reusable library on polynomials.

Regarding polynomial representation, we have used a sparse, normalized and uniform representation. That is, having fixed the number of variables, a canonical form can be associated to each polynomial. In this canonical representation all monomials are arranged in a strictly decreasing order, there are no null monomials and all of them have the same number of variables. The main advantage of this representation arises when deciding equality [9].

Monomial lists are used as the internal representation of polynomials. Monomials are also lists consisting of a coefficient and a term. Having selected a set of variables and an ordering on them, each term is uniquely represented by a list of natural numbers. Although most of the theory is done for an arbitrary field, via the encapsulation principle, we use polynomials over the field of rational numbers for our implementation of Buchberger's algorithm. This alleviates some proofs at the cost of some generality, as ACL2 can use its built-in linear arithmetic decision procedure. In any case, the general theory has to be eventually instantiated to obtain an executable algorithm.

The functions `k-polynomialp` and `k-polynomialsp` recognize polynomials and polynomial lists (with `k` variables and rational coefficients). Analogously, `+`, `*`, `-` and `|0|` stand for polynomial addition, multiplication, negation and the zero polynomial. Let us now introduce the notion of ideal, along with the formalization of polynomial ideals in ACL2.

Definition 1. *$I \subseteq R$ is an ideal of R if it is closed under addition and under the product by elements of R.*

Definition 2. *The ideal generated by $B \subseteq R$, denoted as $\langle B \rangle$, is the set of linear combinations of B with coefficients in R. We say that B is a basis of $I \subseteq R$ if $I = \langle B \rangle$. An ideal is finitely-generated if it has a finite basis.*

Hilbert's Basis Theorem implies that every ideal in $K[x_1, \ldots, x_k]$ is finitely-generated, if K is a field. Polynomial ideals can be expressed in ACL2 by taking this into account. Let C and F be lists of polynomials. The predicate $p \in \langle F \rangle$ can be restated as $\exists C\ p = \mathit{lin\text{-}comb}(C, F)$, where *lin-comb* is a recursive function computing the linear combination of the elements in F with coefficients in C.

As ACL2 is a quantifier-free logic we use a common trick: we introduce a Skolem function assumed to return a list of coefficients witnessing the ideal membership. In ACL2 this can be expressed in the following way:

```
(defun-sk<k> in<> (p F)
  (exists (C) (and (k-polynomialsp C) (equal p (lin-comb C F)))))
```

The use of `exists` in this definition is just syntactic sugar. Roughly speaking, the above construction introduces a Skolem function `in<>-witness`, with arguments `p` and `F`, which is axiomatized to choose, if possible, a list `C` of polynomial coefficients such that when linearly combined with the polynomials in `F`, `p` is obtained. Thus, `C` is a witness of the membership of `p` to the ideal generated by `F`, and `in<>` is defined by means of `in<>-witness`. The following theorems establish that our definition of ideal in ACL2 meets the intended closure properties:

```
(defthm |p in <F> & q in <F> => p + q in <F>|
  (implies (and (k-polynomialp p) (k-polynomialp q) (k-polynomialsp F))
           (implies (and (in<> p F) (in<> q F)) (in<> (+ p q) F))))

(defthm |q in <F> => p * q in <F>|
  (implies (and (k-polynomialp p) (k-polynomialp q) (k-polynomialsp F))
           (implies (in<> q F) (in<> (* p q) F))))
```

Whenever a theorem about `in<>` is proved we have to provide ACL2 with a hint to construct the necessary witness. For example, to prove that polynomial ideals are closed under addition we straightforwardly built an intermediate function computing the witness of $p+q \in \langle F \rangle$ from those of $p \in \langle F \rangle$ and $q \in \langle F \rangle$.

Definition 3. *The congruence induced by an ideal I, written as $\equiv_I$, is defined by $p \equiv_I q \iff p - q \in I$.*

The definition of $\equiv_{\langle F \rangle}$ in ACL2 is immediate[3]:

```
(defun<k> =<> (p q F)
  (in<> (+ p (- q)) F))
```

Clearly, the ideal membership problem for an ideal I is solvable if, and only if, its induced congruence $\equiv_I$ is decidable. Polynomial reductions will help us to design decision procedures for that congruence.

4 Polynomial Reductions

Let $<_M$ be a well-founded ordering on monomials, $p \neq 0$ a polynomial and let $lm(p)$ denote the leader monomial of p with respect to $<_M$.

Definition 4. *Let $f \neq 0$ be a polynomial. The reduction relation on polynomials induced by f, denoted as $\rightarrow_f$, is defined such that $p \rightarrow_f q$ if p contains a monomial $m \neq 0$ such that there exists a monomial c such that $m = -c \cdot lm(f)$ and $q = p + c \cdot f$. If $F = \{f_1, \ldots, f_k\}$ is a finite set of polynomials, then the reduction relation induced by F is defined as $\rightarrow_F \; = \bigcup_{i=1}^{k} \rightarrow_{f_i}$.*

[3] For the sake of readability, we use `defun-sk<k>` and `defun<k>`, instead of `defun-sk` and `defun`. These are just macros which add an extra parameter `k` (the number of variables) to a function definition, so we do not have to specify it in each function application. When `k` is not involved, `defun` is used.

We have formalized polynomial reductions in the framework of abstract reductions developed in [11]. This approach will allow us to export, by functional instantiation, well-known properties of abstract reductions (for example, Newman's lemma) to the case of polynomial reductions, avoiding the need to prove them from scratch.

In [11], instead of defining reductions as binary relations, they are defined as the action of *operators* on elements, obtaining reduced elements. More precisely, the representation of a reduction relation requires defining three functions:

1. A unary predicate specifying the domain where the reduction is defined. In our case, polynomials, as defined by the function `k-polynomialp`.
2. A binary function, `reduction`, computing the application of an operator to a polynomial. In our case, operators are represented by structures $\langle m, c, f\rangle$ consisting of the monomials m and c, and the polynomial f appearing in the definition of the polynomial reduction relation (Def. 4).
3. A binary predicate checking whether the application of a given operator to a given object is *valid*. The application of an operator $\langle m, c, f\rangle$ to p is valid if p is a polynomial containing the monomial m, $f \neq 0$ is a polynomial in F and $c = -m/lm(f)$. Notice that the last requirement implies that $lm(f)$ must divide m. This validity predicate is implemented by a function `validp`.

These functions are just what we need to define in ACL2 all the concepts related to polynomial reductions. Let us begin defining $\leftrightarrow_F$ (the symmetric closure of $\rightarrow_F$). We need the notion of *proof step* to represent the connection of two polynomials by the reduction relation, in either direction (direct or inverse). Each proof step is a structure consisting of four fields: a boolean field marking the step direction, the operator applied, and the elements connected (`elt1`, `elt2`). A proof step is *valid* if one of its elements is obtained by a valid application of its operator to the other element in the specified direction. The function `valid-proof-stepp` (omitted here), checks the validity of a proof step.

The following function formalizes in ACL2 the relation $\leftrightarrow_F$. Note that due to the absence of existential quantification, the `step` argument is needed to explicitly introduce the proof step justifying that $p \leftrightarrow_F q$.

```
(defun <-> (p q step F)
  (and (valid-proof-stepp step F)
       (equal p (elt1 step)) (equal q (elt2 step))))
```

Next, we define $\overset{*}{\leftrightarrow}_F$ (the equivalence closure of $\rightarrow_F$). This can be described by means of a sequence of concatenated proof steps, which we call a *proof*[4]. Note that again due to the absence of existential quantification, the `proof` argument explicitly introduces the proof steps justifying that $p \overset{*}{\leftrightarrow}_F q$.

```
(defun <->* (p q proof F)
  (if (endp proof)
```

[4] Notice that the meaning of the word "proof" here is different than in the expression "ACL2 proof". This proof is just a sequence of reduction steps. In fact, we are formalizing an algebraic proof system inside ACL2.

```
      (and (equal p q) (k-polynomialp p))
    (and (k-polynomialp p)
         (<-> p (elt2 (first proof)) (first proof) F)
         (<->* (elt2 (first proof)) q (rest proof) F))))
```

In the same way, we define the relation $\rightarrow_F^*$ (the transitive closure of $\rightarrow_F$), by a function called `->*` (in this case, we also check that all proof steps are direct).

The following theorems establish that the congruence $\equiv_{\langle F \rangle}$ is equal to the equivalence closure $\overset{*}{\leftrightarrow}_F$. This result is crucial to connect the results about reduction relations to polynomial ideals.

```
(defthm |p =<F> q => p <->F* q|
  (let ((proof (|p =<F> q => p <->F* q|-proof p q F)))
    (implies (and (k-polynomialp p) (k-polynomialp q) (k-polynomialsp F)
                  (=<> p q F))
             (<->* p q proof F)))

(defthm |p <->F* q => p =<F> q|
  (implies (and (k-polynomialp p) (k-polynomialp q) (k-polynomialsp F)
                (<->* p q proof F))
           (=<> p q F)))
```

These two theorems establish that it is possible to obtain a sequence of proof steps justifying that $p \overset{*}{\leftrightarrow}_F q$ from a list of coefficients justifying that $p - q \in \langle F \rangle$, and vice versa. The expression `(|p =<F> q => p <->F* q|-proof p q F)` explicitly computes such proof, in a recursive way. This is typical in our development: in many subsequent ACL2 theorems, the `proof` argument in `<->*` or `->*` will be locally-bound (through a `let` or `let*` form) to a function computing the necessary proof steps. As these functions are rather technical and it would take long to explain them, we will omit their definitions. But it is important to remark this constructive aspect of our formalization.

Next, we proceed to prove the Noetherianity of the reduction relation. In the sequel, `<` represents the polynomial ordering whose well-foundedness was proved in [9][5]. This ordering can be used to state the Noetherianity of the polynomial reduction. For this purpose, it suffices to prove that the application of a valid operator to a polynomial produces a smaller polynomial with respect to this well-founded relation:

```
(defthm |validp(p, o, F) => reduction(p, o) < p|
  (implies (and (k-polynomialp p) (k-polynomialsp F))
           (implies (validp p o F) (< (reduction p o) p))))
```

As a consequence of Noetherianity we can define the notion of normal form.

Definition 5. *A polynomial p is in normal form or is irreducible w.r.t. $\rightarrow_F$ if there is no q such that $p \rightarrow_F q$. Otherwise, p is said to be reducible. A polynomial q is a normal form of p w.r.t. $\rightarrow_F$ if $p \rightarrow_F^* q$ and q is irreducible w.r.t. $\rightarrow_F$.*

[5] As it is customary in ACL2, this is proved by means of an ordinal embedding into ϵ_0.

The notion of normal form of a polynomial can be easily defined in our framework. First, we define a function `reducible`, implementing a reducibility test: when applied to a polynomial `p` and to a list of polynomials `F`, it returns a valid operator, whenever it exists, or `nil` otherwise. The following theorems state the main properties of `reducible`:

```
(defthm |reducible(p, F) => validp(p, reducible(p, F), F)|
  (implies (reducible p F)
           (validp p (reducible p F) F)))

(defthm |~reducible(p, F) => ~validp(p, o, F)|
  (implies (not (reducible p F))
           (not (validp p o F))))
```

Now it is easy to define a function `nf` that computes a normal form of a given polynomial with respect to the reduction relation induced by a given list of polynomials. This function is simple: it iteratively tests reducibility and applies valid operators until an irreducible polynomial is found. Note that termination is guaranteed by the Noetherianity of the reduction relation and the well-foundedness of the polynomial ordering.

```
(defun<k> nf (p F)
  (if (and (k-polynomialp p) (k-polynomialsp F))
      (let ((red (reducible p F)))
        (if red (nf (reduction p red) F) p))
    p))
```

The following theorems establish that, in fact, `nf` computes normal forms. Again, in order to prove that $p \rightarrow_F^* nf_F(p)$, we have to explicitly define a function `|p ->F* nf(p, F)|-proof` which construct a proof justifying this. This function is easily defined by collecting the operators returned by `reducible`.

```
(defthm |p ->F* nf(p, F)|
  (let ((proof (|p ->F* nf(p, F)|-proof p F)))
    (implies (and (k-polynomialp p) (k-polynomialsp F))
             (->* p (nf p F) proof F))))

(defthm |nf(p, F) irreducible|
  (implies (and (k-polynomialp p) (k-polynomialsp F))
           (not (validp (nf p F) o F))))
```

Although `nf` is suitable for reasoning about normal form computation, it is not suitable for being used by an implementation of Buchberger's algorithm: for example, `nf` explicitly deals with operators, which are a concept of theoretical nature. At this point, we talk about the polynomial reduction function red_F^* used in Buchberger's algorithm. This function (whose definition we omit) do not make any use of operators but is modeled from the closure of the set extension of another function, *red*, which takes two polynomials as its input and returns the result of reducing the first polynomial with respect to the second one. The following theorem shows the equivalence between nf_F and red_F^*:

```
(defthm |nf(p, F) = red*(p, F)|
  (implies (and (k-polynomialp p) (k-polynomialsp F))
           (equal (nf p F) (red* p F))))
```

With this result, we can translate all the properties proved about `nf` to `red*`. This is typical in our formalization: we use some functions for reasoning, and other functions for computing, translating the properties from one to another by proving equivalence theorems. For example, we proved the stability of the ideal with respect to `red*` using this technique.

5 Gröbner Bases

The computation of normal forms with respect to a given ideal can be seen as a generalized polynomial division algorithm, and the normal form computed as the "remainder" of that division. The ideal membership problem can be solved taking this into account: compute the normal form and check for the zero polynomial. Unfortunately, it is possible that, for a given basis F, a polynomial in $\langle F \rangle$ cannot be reduced to the zero polynomial. This is where Gröbner bases come into play:

Definition 6. *G is a Gröbner basis of the ideal generated by F if $\langle G \rangle = \langle F \rangle$ and $p \in \langle G \rangle \iff p \rightarrow_G^* 0$.*

The key point in Buchberger's algorithm is that the property of being a Gröbner basis can be deduced by only checking that a finite number of polynomials (called *s-polynomials*) are reduced to zero:

Definition 7. *Let p and q be polynomials. Let m, m_1 and m_2 be monomials such that $m = \mathrm{lcm}(lm(p), lm(q))$ and $m_1 \cdot lm(p) = m = m_2 \cdot lm(q)$. The s-polynomial induced by p and q is defined as $s\text{-}poly(p, q) = m_1 \cdot p - m_2 \cdot q$*

Theorem 1. *Let $\Phi(F) \equiv \forall p, q \in F \ s\text{-}poly(p, q) \rightarrow_F^* 0$. The reduction induced by F is locally confluent if $\Phi(F)$ is verified. That is:*

$$\Phi(F) \implies \forall p, q, r \ (r \rightarrow_F p \wedge r \rightarrow_F q \implies \exists s \ (p \rightarrow_F^* s \wedge q \rightarrow_F^* s))$$

This theorem was the most difficult to formalize and prove in our work. First, note that it cannot be stated as a single theorem in the quantifier-free ACL2 logic, due to the universal quantifier in its hypothesis, $\Phi(F)$. For this reason, we state its hypothesis by the following `encapsulate` (we omit the local witnesses and some nonessential technical details):

```
(encapsulate
 ((F () t) (s-polynomial-proof (p q) t))
  ...
 (defthm |Phi(F)|
   (let ((proof (s-polynomial-proof p q)))
     (and (k-polynomials (F))
          (implies (and (in<> p (F)) (in<> q (F)))
                   (->* (s-poly p q) (|0|) proof (F)))))))
```

The first line of this `encapsulate` presents the *signature* of the functions it introduces, and the theorem inside can be seen as an assumed property about these functions. In this case, we are assuming that we have a list of polynomials given by the 0-ary function `F`, with the property that every s-polynomial formed with pairs of elements of `(F)` is reduced to `(|0|)`. This reduction is justified by a function `s-polynomial-proof` computing the corresponding sequence of proof steps representing the reduction to `(|0|)`. We insist that `F` and `s-polynomial-proof` are not completely defined: we are only assuming `|Phi(F)|` about them.

Now, the conclusion of Th. 1 is established as follows:

```
(defthm |Phi(F) => local-confluence(->F)|
  (let ((proof2 (transform-local-peak-F proof1)))
    (implies (and (k-polynomial p) (k-polynomial q)
                  (<->* p q proof1 (F)) (local-peakp proof1))
             (and (<->* p q proof2 (F)) (valleyp proof2)))))
```

This theorem needs some explanation. Note that local confluence can be reformulated in terms of the "shape" of the involved proofs: a reduction is locally confluent if, and only if, for every local peak proof (that is, of the form $p \leftarrow r \rightarrow q$) there exists an equivalent valley proof (that is, of the form $p \stackrel{*}{\rightarrow} s \stackrel{*}{\leftarrow} q$). It is easy to define in ACL2 the functions `local-peakp` and `valleyp`, checking those shapes of proofs. Note that again due to the absence of existential quantification, the valley proof in the above theorem is given by a function `transform-local-peak-F`, such that from a given local peak proof, it computes an equivalent valley proof. The definition of this function is very long and follows the same case distinction as in the classical proof of this result; only in one of its cases (the one dealing with "overlaps"), `s-polynomial-proof` is used as an auxiliary function, reflecting in this way where the assumption about $\Phi(F)$ is necessary.

The last step in this section follows from general results of abstract reduction relations. In particular, if a reduction is locally confluent and Noetherian then its induced equivalence can be decided by checking if normal forms are equal. This has been proved in ACL2 [11] as a consequence of Newman's lemma, also proved there. We can reuse this general result by functional instantiation and obtain an ACL2 proof of the fact that, if $\Phi(F)$, $p \stackrel{*}{\leftrightarrow}_F q \iff nf_F(p) = nf_F(q)$.

With this result, and using the equality between nf_F and red^*_F, and the equality between $\equiv_{\langle F \rangle}$ and $\stackrel{*}{\leftrightarrow}_F$, it can be easily deduced that if $\Phi(F)$ then F is a Gröbner basis (of $\langle F \rangle$). This is established by the following theorem (notice that `(F)` is still the list of polynomials assumed to have property Φ by the above `encapsulate`):

```
(defthm |Phi(F) => (p in <F> <=> red*(p, F) = 0)|
    (implies (k-polynomial p)
             (iff (in<> p (F)) (equal (red* p (F)) (|0|)))))
```

6 Buchberger's Algorithm

Buchberger's algorithm obtains a Gröbner basis of a given finite set of polynomials F by the following procedure: if there is a s-polynomial of F such that its

normal form is not zero, then this normal form can be added to the basis. This makes it reducible to zero (without changing the ideal), but new s-polynomials are introduced that have to be checked. This *completion* process is iterated until all the s-polynomials of the current basis are reducible to zero.

In order to formalize an executable implementation of Buchberger's algorithm in ACL2, several helper functions are needed. The function `initial-pairs` returns all the ordered pairs from the elements of a list. The main function computes the initial pairs from a basis and starts the real computation process.

```
(defun Buchberger (F)
  (Buchberger-aux F (initial-pairs F)))
```

Next, the function that computes a Gröbner basis from an initial basis is defined. This function takes the initial basis and a list of pairs as its input. The function `pairs` returns the ordered pairs built from its first argument and every element in its second argument. As all ACL2 functions must be total and we need to deal with polynomials with a fixed set of variables to ensure termination of the function, we have to explicitly check that the arguments remain in the correct domain. We will comment more about these "type conditions" in Sect. 7.

```
(defun<k> Buchberger-aux (F C)
  (if (and (naturalp k) (k-polynomialsp F) (k-polynomial-pairsp C))
      (if (endp C)
          F
        (let* ((p (first (first C))) (q (second (first C)))
               (h (red* (s-poly p q) F)))
          (if (equal h (|0|))
              (Buchberger-aux F (rest C))
            (Buchberger-aux (cons h F) (append (pairs h F) (rest C))))))
    F))
```

A measure has to be supplied to prove the termination of the above function, so that it can be admitted by the principle of definition. The following section explains this issue.

6.1 Termination

Termination of Buchberger's algorithm can be proved using a lexicographic measure on its arguments. This is justified by the following observations:

1. In the first recursive branch, the first argument keeps unmodified while the second argument structurally decreases since one of its elements is removed.
2. In the second recursive branch, the first argument decreases in a certain well-founded sense despite of the inclusion of a new polynomial. This is a consequence of Dickson's lemma.

Lemma 1 (Dickson). *Let $k \in \mathbb{N}$ and $m_1, m_2, \ldots$ an infinite sequence of monomials with k variables. Then, there exist indices $i < j$ such that m_i divides m_j.*

If we consider the sequence of terms consisting of the leader terms of the polynomials added to the first argument, Dickson's lemma implies termination of Buchberger's algorithm. This is because the polynomial added to the basis, h, is not 0 and it cannot be reduced by F. Consequently, its leader term is not divisible by *any* of the leader terms of the polynomials in F.

Dickson's lemma has been formalized in ACL2 in [7] and [12]. In both cases it has been proved by providing an ordinal measure on finite sequences of terms such that this measure decreases every time a new term not divisible by any of the previous terms in the sequence is added.

We have defined a measure along these lines to prove the termination of Buchberger's algorithm. In fact, our measure is defined on top of the measures used to prove Dickson's lemma in [7, 12], lexicographically combined with the length of the second argument. Although both proofs of Dickson's lemma are based on totally different ideas, the results obtained can be used interchangeably in our formalization.

6.2 Partial Correctness

In order to show that *Buchberger* computes a Gröbner basis, and taking into account the results of the previous section, we just have to prove that $p \in \langle F \rangle \iff p \in \langle \mathit{Buchberger}(F) \rangle$ and that $\mathit{Buchberger}(F)$ satisfies Φ. The following ACL2 theorems establish these two properties:

```
(defthm |<Buchberger(F)> = <F>|
  (implies (and (k-polynomialp p) (k-polynomialsp F))
           (iff (in<> p (Buchberger F)) (in<> p F))))

(defthm |Phi(Buchberger(F))|
  (let ((G (Buchberger F)) (proof (|Phi(Buchberger(F))|-proof p q F)))
    (implies (and (k-polynomialp p) (k-polynomialp q) (k-polynomialsp F)
                  (in<> p G) (in<> q G))
             (->* (s-poly p q) (|0|) proof G))))
```

The statement of this last theorem deserves some comments. Our ACL2 formulation of Th. 1 defines the property $\Phi(F)$ as the existence of a function such that for every s-polynomial of F, it computes a sequence of proof steps justifying its reduction to `(|0|)` (assumption `|Phi(F)|` in the `encapsulate` of the previous section). Thus, if we want to establish the property Φ for a particular basis (the basis returned by `Buchberger` in this case), we must explicitly define such function and prove that it returns the desired proofs for every s-polynomial of the basis. In this case the function is called `|Phi(Buchberger(F))|-proof`. For the sake of brevity, we omit the definition of this function, but it is very interesting to point out that it is based in a recursion scheme very similar to the recursive definition of `Buchberger-aux`. This function collects, every time a new s-polynomial is examined, the corresponding proof justifying its reduction to the zero polynomial.

6.3 Deciding Ideal Membership

Finally, we can compile the results above to define a decision procedure for ideal membership. This procedure just checks whether a given polynomial reduces to 0 with respect to the Gröbner basis returned by Buchberger's algorithm.

```
(defun<k> imdp (p F)
  (equal (red* p (Buchberger F)) (|0|)))
```

Theorem 2. $G = Buchberger(F) \implies (p \in \langle F \rangle \iff red^*_G(p) = 0)$.

The ACL2 theorem stating the soundness and completeness of the decision procedure follows, as an easy consequence of the correctness of `Buchberger` and the theorem `|Phi(F) => (p in <F> <=> red*(p, F) = 0)|` in Sect. 5.

```
(defthm |p en <F> <=> imdp(p, F)|
    (implies (and (k-polynomialp p) (k-polynomialsp F))
             (iff (in<> p F) (imdp p F))))
```

In this context, the theorem `|Phi(F) => (p in <F> <=> red*(p, F) = 0)|` is used by functional instantiation, replacing `F` by `(lambda () (Buchberger F))` and `s-polynomial-proof` by `|Phi(Buchberger(F))|-proof`.

Note that all the functions used in the definition of the decision procedure are executable and therefore the procedure is also executable. Note also that we do not mention operators or proofs, neither when defining the decision procedure nor when stating its correctness. These are only intermediate concepts, which make reasoning more convenient.

7 Conclusions

We have shown how it is possible to use the ACL2 system in the formal development of Computer Algebra algorithms by presenting a verified implementation of Buchberger's algorithm and a verified decision procedure for the ideal membership problem. It is interesting to point out that all the theory needed to prove the correctness of the algorithm has been developed in the ACL2 logic, in spite of its (apparently) limited expressiveness.

We have benefited from work previously done in the system. In particular, all the results about abstract reductions were originally developed for a formalization of rewriting systems [11]. We believe that this is a good example of how seemingly unrelated formalizations can be reused in other projects, provided the system offers a minimal support for it. However, we feel that ACL2 could be improved to provide more comfortable mechanisms for functional instantiation and for abstraction in general. Encapsulation provides a good abstraction mechanism but functionally instantiating each encapsulated theorem is a tedious and error-prone task. Recently, several proposals have been formulated (e.g. polymorphism and abstract data types) to cope with this problem in ACL2 [6]. A graphical interface to visualize proof trees would be helpful too.

[6] A similar modularity issue has been reported in the COQ system too [13].

Little work has been done on the machine verification of Buchberger's algorithm. As we mentioned in the introduction, the most relevant is the work of L. Théry [13] in Coq. T. Coquand and H. Persson [3] report an incomplete integrated development in Martin-Löf's type theory using Agda. There is also a Mizar project [10] to formalize Gröbner bases. The main difference between our approach and these works is the underlying logic. All these logics are very different from Acl2, which is more primitive, basically an untyped and quantifier-free logic of total recursive functions, and makes no distinction between the programming and the specification languages. In exchange, a high degree of automation can be achieved and executability is obtained for free.

We think that an advantage of our approach is that the implementation presented is compliant with Common Lisp, a real programming language, and can be directly executed in Acl2 or in any compliant Common Lisp. This is not the case of other systems, where the logic is not executable at all or the code has to be extracted by unverified means. Taking into account that Lisp is the language of choice for the implementation of CAS, like Macsyma and Axiom, this is not just a matter of theoretical importance but also a practical one.

Our formal proof differs from Théry's. First, it is based on [1] instead of [4]. Second, we prove that Φ implies local-confluence instead of confluence: compare this with the proof of *SpolyImpConf* in [13]. Differences extend also to definitions, e.g. ideals and confluence, mainly motivated by the lack of existential quantification. Finally, [13] uses a non-constructive proof of Dickson's lemma by L. Pottier[7]. Our termination argument uses a proof of Dickson's lemma obtained by an ordinal embedding in ϵ_0, the only well-founded structure known to Acl2.

We would like to remark that although polynomial properties seem trivial to prove, this is not the case [8, 9]. It seems that this is not due to the simplicity of the Acl2 logic. In [10] the authors recognize that it was challenging to prove the associativity of polynomial multiplication in Mizar, a system devoted to the formalization of mathematics. They were amazed by the fact that, in well-known Algebra treatises, these properties are usually reduced to the univariate case or their proofs are partially sketched and justified "by analogy". In some cases, the proofs are even left as an exercise. In the same way, the author of [13] had to devote a greater effort to polynomials due to problems arising during their formalization in Coq.

As for the user interaction required, we provided 169 definitions and 560 lemmas to develop a theory of polynomials (although this includes more than the strictly needed here) and 109 definitions and 346 lemmas for the theory of Gröbner bases and Buchberger's algorithm. All these lemmas are proved almost automatically. It is worth pointing out that of the 333 lemmas proved by induction, only 24 required a user-supplied induction scheme. Other lemmas needed a hint about the convenience of using a given instance of another lemma or keeping a function definition unexpanded. Only 9 functions required a hint for their termination proofs. Thus, the main role of the user is to provide the suitable sequence of definitions and lemmas to achieve the final correctness theorem.

[7] A new proof of Dickson's lemma in Coq by H. Persson has been proposed later.

Théry's implementation provides some standard optimizations that we do not include. Regarding future work, we are interested in studying how our verified implementation could be improved to incorporate some of the refinements built into the very specialized and optimized (and not formally verified) versions used in industrial-strength applications. It would be also interesting to use it to verify some application of Gröbner bases such as those described in [2].

An obvious improvement in the verified implementation is to avoid the "type conditions" in the body of `Buchberger-aux`, since these conditions are unnecessarily evaluated in every recursive call. But these conditions are needed to ensure termination. Until ACL2 version 2.7, there was no way to avoid this; but since the recent advent of ACL2 version 2.8 that is no longer true, since it is possible for a function to have two different bodies, one used for execution and another for its logical definition: this is done by previously proving that both bodies behave in the same way on the intended domain of the function. We plan to apply this new feature to our definition of Buchberger's algorithm.

References

1. Baader, F. & Nipkow, T.: *Term Rewriting and All That.* Cambridge University Press (1998)
2. Buchberger, B. & Winkler, F. (eds.): *Gröbner Bases and Applications.* London Mathematical Society Series **251** (1998)
3. Coquand, T. & Persson, H.: *Gröbner Bases in Type Theory.* Types for Proofs and Programs, International Workshop. LNCS **1657**:33–46 (1999)
4. Geddes, K. O.; Czapor, S. R. & Labahn, G.: *Algorithms for Computer Algebra.* Kluwer (1998)
5. Kaufmann, M.; Manolios, P. & Moore, J S.: *Computer-Aided Reasoning: An Approach.* Kluwer (2000)
6. Kaufmann, M. & Moore, J S.: *Structured Theory Development for a Mechanized Logic.* Journal of Automated Reasoning **26**(2):161–203 (2001)
7. Martín, F. J.; Alonso, J. A.; Hidalgo, M. J. & Ruiz, J. L.: *A Formal Proof of Dickson's Lemma in ACL2.* Logic for Programming, Artificial Intelligence and Reasoning. LNAI **2850**:49-58 (2003)
8. Medina, I.; Alonso, J. A. & Palomo, F.: *Automatic Verification of Polynomial Rings Fundamental Properties in ACL2.* ACL2 Workshop 2000. Department of Computer Sciences, University of Texas at Austin. TR–00–29 (2000)
9. Medina, I.; Palomo, F. & Alonso, J. A.: *Implementation in ACL2 of Well-Founded Polynomial Orderings.* ACL2 Workshop 2002 (2002)
10. Rudnicki, P.; Schwarzweller, C. & Trybulec, A.: *Commutative Algebra in the Mizar System.* Journal of Symbolic Computation **32**(1–2):143–169 (2001)
11. Ruiz, J. L.; Alonso, J. A.; Hidalgo, M. J. & Martín, F. J.: *Formal Proofs about Rewriting using ACL2.* Annals of Mathematics and Artificial Intelligence **36**(3): 239–262 (2002)
12. Sustyk, M.: *Proof of Dickson's Lemma Using the ACL2 Theorem Prover via an Explicit Ordinal Mapping.* Fourth International Workshop on the ACL2 Theorem Prover and Its Applications (2003)
13. Théry, L.: *A Machine-Checked Implementation of Buchberger's Algorithm.* Journal of Automated Reasoning **26**(2): 107–137 (2001)