A Certified Version of Buchberger's Algorithm

Laurent Théry

INRIA, 2004 route des Lucioles, 06902 Sophia Antipolis France
thery@sophia.inria.fr

Abstract. We present a proof of Buchberger's algorithm that has been developed in the Coq proof assistant. The formulation of the algorithm in Coq can then be efficiently compiled and used to do computation.

1 Introduction

If we look at the way one can use computers to do mathematics, there is a clear separation between computing, where one uses computer algebra systems, and proving, where one uses theorem provers. The fact that these two aspects are covered separately has obvious drawbacks. On the one hand it is a well-known fact that, because of misuse or implementation errors, one should always double-check the results given by a computer algebra system. This problem is even more crucial for general-purpose computer algebra systems where the library of algorithms is mostly developed by the user community. Extensions of the system are then performed without giving evidence of their applicability or correctness. On the other hand theorem provers usually come with very little computing power. This makes it difficult to complete proofs for which some computing steps are needed.

It would be a real progress if one could unify these two aspects in a single system. It would then be possible to define mathematical objects and both compute and prove properties about them. Building such a system from scratch requires an important effort. A more pragmatic approach consists in complementing existing systems. If we look at computer algebra systems, the situation is somewhat difficult. The languages of general-purpose computer algebra systems have not been designed with the idea that people would like to reason about them. For example, the scope of local variables in Maple [2] is not limited to the procedure where they have been defined. Thus, stating properties of algorithms turns out to be very difficult.

If we look at theorem provers, the main problem is efficiency. While most theorem provers allow us to define algorithms, executing them is inefficient because it is performed inside the prover in an interpretative way. An alternative solution is for the prover to be able to translate its algorithms into another programming language that has a compiler.

Our approach follows the second line. We have chosen the theorem prover Coq [12] to do our experiments. Coq is a prover based on type theory. It manipulates objects with a rich notion of types which is clearly adequate for mathematical objects. Coq also proposes an extraction mechanism that, given an algorithm defined in the system, generates an implementation in the language Ocaml [14] that can be efficiently compiled.

Is this solution practical? What is the effort involved in trying to certify standard algorithms for computer algebra systems? It is to answer these questions that we decided to work on the proof of correctness of Buchberger's algorithm. We started from a five page description of the algorithm in a standard introduction book [7]. The goal was simple: to develop enough mathematical knowledge in Coq for stating the algorithm and proving its correctness and termination.

The paper is organized as follows. In Section 2 we introduce the Buchberger's algorithm. In Sections 3 and 4 we sketch its proofs of correctness and termination. In Section 5 we explain the main steps of our development and give a running example of the algorithm. Finally we relate our approach to others and draw some conclusions and future work.

2 Buchberger's Algorithm

Buchberger's algorithm is a *completion* algorithm working on polynomials. Given a list of polynomials it returns a completed list that has a particular property. Before presenting the algorithm, we first need to define some basic notions [7].

2.1 Ordered Polynomials

We first consider the usual n variables polynomials over an arbitrary field $(A, +_a, -_a, *_a, /_a, 0_a, 1_a)$ with two of their usual operations: addition $(+)$ and multiplication by a term $(.)$. A polynomial is composed of a list of *terms*. Each term is composed of a *coefficient* and a *monomial*. The set of coefficients is A. The set of monomials is denoted by M_n where n is the number of variables. The set of terms and polynomials are denoted by T_{A,M_n} and P_{A,M_n} respectively.

An *order* $\leq_{M_n}$ over monomials is a binary relation that is transitive, reflexive and antisymmetric. It is *total* if two distinct elements are always comparable. It is *well-founded* if there exists no infinite strictly decreasing sequence of monomials. Finally it is *admissible* if $x_1^0 \dots x_n^0$ is minimal for the order and if the order is compatible with the multiplication.

Given an admissible well-founded total order $\leq_{M_n}$ over monomials, it is possible to represent a polynomial as a list of terms, such that the list of the corresponding monomials is ordered, i.e. each monomial in the list is strictly greater than the ones at its right. We use 0 and $\overset{>}{+}$ to denote the null polynomial and the ordered list constructor respectively. From this representation we get the structural induction theorem for an arbitrary predicate P over polynomials:

$$(P\,0) \wedge (\forall a \in A, \forall p \in P_{A,M_n}, (P\,p) \Rightarrow (P\,(a \overset{>}{+} p))) \Rightarrow \forall p \in P_{A,M_n}, (P\,p)$$

We define the transitive relation $<_p$ over polynomials as the smallest relation such that:

- $\forall t \in T_{A,M_n}, \forall p \in P_{A,M_n}, 0 <_p t \vec{+} p$
- $\forall a_1, a_2 \in A, \forall m \in M_n, \forall p, q \in P_{A,M_n}, p <_p q \Rightarrow a_1 m \vec{+} p <_p a_2 m \vec{+} q$
- $\forall a_1, a_2 \in A, \forall m_1, m_2 \in M_n, \forall p, q \in P_{A,M_n},$
  $m_1 <_{M_n} m_2 \Rightarrow a_1 m_1 \vec{+} p <_p a_2 m_2 \vec{+} q$

This relation is well-founded. So we get another important induction principle:

$$(\forall p \in P_{A,M_n}, (\forall q \in P_{A,M_n}, q <_p p \Rightarrow (P\, q)) \Rightarrow (P\, p)) \Rightarrow \forall p \in P_{A,M_n}, (P\, p)$$

2.2 Normal Form

Given the definition of polynomials, it is possible that polynomials carry terms with null coefficient. Equality for polynomials is then understood as the equality without paying attention to terms with null coefficient. To give a more algorithmic account of this notion, we define the function *nf* that computes the normal form of a polynomial by removing terms with null coefficient:

- $nf(0) = 0;$
- $\forall m \in M_n, \forall p \in P_{A,M_n}, nf(0_a m \vec{+} p) = nf(p);$
- $\forall a \in A, \forall m \in M_n, \forall p \in P_{A,M_n}, a \neq 0_a \Rightarrow nf(am \vec{+} p) = am \vec{+} nf(p).$

2.3 One Step Division, Reduction, and Irreducibility

Since the division over monomials is not total, we first define a relation $divP_{M_n}$:

$$\forall m_1, m_2 \in M_n,\ divP_{M_n}(m_1, m_2) \iff \exists m_3 \in M_n,\ m_1 = m_3.m_2$$

Then the division over monomials $/_{M_n}$ is defined as:

$$\forall m_1, m_2 \in M_n,\ divP_{M_n}(m_1, m_2) \Rightarrow m_1 = (m_1 /_{M_n} m_2).m_2$$

We define the one step division $/_p$ over polynomials as follows :

$$\begin{aligned}
&\forall m_1, m_2 \in M_n,\ divP_{M_n}(m_1, m_2) \Rightarrow \\
&\forall a_1, a_2 \in A,\ \forall p_1, p_2 \in P_{A,M_n},\ a_2 \neq 0_a \Rightarrow \\
&\qquad (a_1 m_1 \vec{+} p_1) /_p (a_2 m_2 \vec{+} p_2) = p_1 - (a_1 /_a a_2)(m_1 /_{M_n} m_2).p_2
\end{aligned}$$

Given a set of polynomials S, it is now possible to define the reduction relation $\rightarrow_S$ as the smallest relation such that:

- $\forall p_1, p_2 \in P_{A,M_n}, \forall t \in T_{A,M_n}, p_1 \rightarrow_S p_2 \Rightarrow t \vec{+} p_1 \rightarrow_S t \vec{+} p_2$
- $\forall m_1, m_2 \in M_n,\ divP_{M_n}(m_1, m_2) \Rightarrow$
  $\forall a_1, a_2 \in A, \forall p_1, p_2 \in P_{A,M_n}, a_2 \neq 0_a \Rightarrow$
  $(a_2 m_2 \vec{+} p_2) \in S \Rightarrow a_1 m_1 \vec{+} p_1 \rightarrow_S (a_1 m_1 \vec{+} p_1) /_p (a_2 m_2 \vec{+} p_2)$

We say that p is irreducible by $\rightarrow_S$ if $\forall q \in P_{A,M_n}, \neg(p \rightarrow_S q)$. We define the relation $\rightarrow_S^+$ as the reflexive-transitive closure of the relation $\rightarrow_S$ and the reduction till irreducibility $\rightarrow_S^*$ ($p \rightarrow_S^* q$ iff $p \rightarrow_S^+ q$ and q is irreducible by $\rightarrow_S$).

2.4 Spolynomials

We use the infix symbol ^ to denote the function that computes the least common multiple of two monomials. If we have two polynomials $p = a_1 m_1 \dot{+} p_1$ and $q = a_2 m_2 \dot{+} p_2$ with $a_1, a_2 \neq 0$, the polynomial $m_1\hat{\ }m_2$ represents the 'smallest' polynomial that can be divided by both polynomials p and q:

$$m_1\hat{\ }m_2 \rightarrow_{\{p\}} q_1 = -(1_a /_a a_1)((m_1\hat{\ }m_2) /_{M_n} m_1).p_1$$

$$m_1\hat{\ }m_2 \rightarrow_{\{q\}} q_2 = -(1_a /_a a_2)((m_1\hat{\ }m_2) /_{M_n} m_2).p_2$$

We define the function *Spoly* as $\mathit{Spoly}(p, q) = q_2 - q_1$ if the previous conditions on p and q hold and $\mathit{Spoly}(p, q) = 0$ otherwise ($p = 0$ or $q = 0$ or $a_1 = 0$ or $a_2 = 0$).

2.5 Polynomial Ideals

A polynomial *ideal* is a set of polynomials I that is stable under

- addition: $\forall p, q \in I,\ p + q \in I$
- multiplication by a term: $\forall p \in I,\ \forall t \in T_{A,M_n},\ t.p \in I$.

Given a set of polynomials S, the ideal $<S>$ *generated by* S is the set of polynomials p such that

$$\exists k \in \mathbb{N}, p = \sum_{i<k} t_i.p_i \text{ such that } \forall i < k, t_i \in T_{A,M_n} \text{ and } p_i \in S.$$

It is easy to check that this set is an ideal. Finally a set of polynomials S is said to be a *basis* of an ideal I iff $<S> = I$.

2.6 Gröbner Basis and Buchberger's Algorithm

To be able to decide whether or not a given polynomial belongs to an ideal is an important property that can be used to solve a large number of interesting problems concerning polynomials. We say that a set of polynomials S is a *Gröbner basis* iff

$$\forall p \in P_{A,M_n},\ p \in <S> \iff p \rightarrow_S^* 0$$

In other words, a Gröbner basis is characterized by a generated ideal whose only irreducible polynomial is 0. Thus, to check if a given polynomial belongs to an ideal generated by a Gröbner basis, one simply needs to reduce it to an irreducible polynomial and then check if this polynomial is 0 or not. A general result by Hironaka states that, given any ideal generated by a set of polynomials, there exists a Gröbner basis that generates the same ideal. Buchberger's contribution was to give an explicit algorithm for computing a Gröbner basis corresponding to the initial set of polynomials.

In the presentation of the algorithm below, we manipulate sets of polynomials as lists. The set of set of polynomials is represented by P_{A,M_n} *list*. We also use []

to denote the empty list and the notation $[p|L]$ to represent the list whose head is the polynomial p and whose tail is L.

We first define the function *SpolyL* that takes a polynomial and two lists of polynomials and returns a list of polynomials:

- $SpolyL(p, L_1, []) = L_1$
- $SpolyL(p, L_1, [q|L_2]) = [Spoly(p, q)|SpolyL(p, L_1, L_2)]$.

This function simply adds to the first list the spolynomials formed by the polynomial and each polynomial of the second list.

The second function *SpolyProd* computes a reduced set of all possible spolynomials formed from a list of polynomials:

- $SpolyProd([]) = []$
- $SpolyProd([p|L]) = SpolyL(p, SpolyProd(L), L)$.

The third function *nfL* normalizes each element of a list, removing zero polynomials:

- $nfL([]) = []$
- $nf(p) \neq 0 \Rightarrow nfL([p|L]) = [nf(p)|nfL(L)]$
- $nf(p) = 0 \Rightarrow nfL([p|L]) = nfL(L)$.

We have now enough material to present the algorithm. Among its parameters there is an arbitrary function *reducef* that takes a polynomial and computes an irreducible polynomial such that:

$$\forall p \in P_{A,M_n} p \rightarrow_S^* reducef(S, p)$$

For the moment, we assume that such a function exists. The algorithm is a completion that takes a pair of lists of polynomials as argument. The first element of the pair represents the basis and the second one the possible candidates to complete the basis:

- $buchf(L_1, []) = L_1$
- $nf(reducef(L_1, p)) \neq 0 \Rightarrow$
  $buchf(L_1, [p|L_2]) = buchf([nf(reducef(L_1, p))|L_1],$
  $SpolyL(nf(reducef(L_1, p)), L_2, L_1))$
- $nf(reducef(L_1, p)) = 0 \Rightarrow buchf(L_1, [p|L_2]) = buchf(L_1, L_2)$.

If the list of candidates is empty, the basis is returned (first case). If the head of the list of candidates does not reduce to zero, it is added to the basis and the spolynomials computed by *SpolyL* are added to the list of candidates (second case). If the head of the list of the candidates reduces to zero, a recursive call is made with the tail of the list (third case).

We finally define the function *buch* that takes a list of polynomials as argument and returns a corresponding Gröbner basis as:

$buch(L) = buchf(nfL(L), SpolyProd(nfL(L)))$.

3 The Proof of Correctness

The correctness of the algorithm can be expressed by two theorems. The first one ensures that the result of the algorithm does not change the generated ideal:
Theorem *BuchStable:*

$$\forall S \in P_{A,M_n} \mathit{list},\ <S>=<\mathit{buch}(S)>$$

The second one states that every member of the ideal reduces to 0:
Theorem *BuchReduce:*

$$\forall S \in P_{A,M_n} \mathit{list},\ \forall p \in <S>,\ p \rightarrow^{*}_{\mathit{buch}(S)} 0$$

The theorem *BuchStable* is a direct consequence of the three following lemmas:
Lemma *RedStable:*

$$\forall S \in P_{A,M_n} \mathit{list},\ \forall p,q \in P_{A,M_n},\ p \rightarrow^{+}_{S} q \Rightarrow (p \in <S> \iff q \in <S>)$$

Lemma *NfStable:*

$$\forall S \in P_{A,M_n} \mathit{list},\ \forall p \in P_{A,M_n},\ (p \in <S> \iff \mathit{nf}(p) \in <S>)$$

Lemma *SpolyStable:*

$$\forall S \in P_{A,M_n} \mathit{list},\ \forall p,q \in P_{A,M_n},\ p \in <S> \wedge q \in <S> \Rightarrow \mathit{Spoly}(p,q) \in <S>$$

The theorem *BuchReduce* needs much more work to be proved. The first step is to prove the three following lemmas:
Lemma *RedCompMinus:*

$$\forall S \in P_{A,M_n} \mathit{list},\ \forall p,q,r \in P_{A,M_n},$$
$$p - q \rightarrow_{S} r \Rightarrow \exists p_1, q_1 \in P_{A,M_n},\ p \rightarrow^{+}_{S} p_1 \wedge q \rightarrow^{+}_{S} q_1 \wedge r = p_1 - q_1$$

Lemma *Red^{+}Minus0*:

$$\forall S \in P_{A,M_n} \mathit{list},\ \forall p,q \in P_{A,M_n},$$
$$p - q \rightarrow^{+}_{S} 0 \Rightarrow \exists r \in P_{A,M_n},\ p \rightarrow^{+}_{S} r \wedge q \rightarrow^{+}_{S} r$$

Lemma *RedDistMinus:*

$$\forall S \in P_{A,M_n} \mathit{list},\ \forall p,q,r \in P_{A,M_n},$$
$$p \rightarrow_{S} q \Rightarrow \exists s \in P_{A,M_n},\ p - r \rightarrow^{+}_{S} s \wedge q - r \rightarrow^{+}_{S} s$$

To prove the first lemma we just look at the term that has been reduced in $p - q$ and use associative and distributive properties of addition and multiplication by a term. The second lemma is proved by induction on the length of the reduction using the first lemma in the induction case. The third lemma is proved with techniques similar to the first one.

The next step is to show that in order to get the theorem *BuchReduce* it is sufficient to prove that the reduction is confluent:

$$\forall p,q,r \in P_{A,M_n},\ (p \rightarrow^{*}_{S} q \wedge p \rightarrow^{*}_{S} r) \Rightarrow q = r$$

Here is the proof:

- We take an arbitrary element p of $<S>$. We want to prove that $p \rightarrow_S^* 0$ with the hypothesis that the reduction is confluent.
- By definition $p = \sum_{i<k} t_i.p_i$ with $\forall i < k, t_i \in T_{A,M_n}$ and $p_i \in S$ for some k.
- We proceed by induction on k.
- For $k = 0$, we have $p = 0$ so the property holds.
- Suppose that the property holds for $l < k$.
- By defining $q = \sum_{i<k-1} t_i.p_i$, we get $q \rightarrow_S^* 0$ by induction hypothesis.
- We have $p - q = t_k p_k$, with $p_k \in S$. It implies that $p - q \rightarrow_S^+ 0$.
- By applying the lemma *Red⁺MinusO*, we deduce that there exists an r such that $p \rightarrow_S^+ r$ and $q \rightarrow_S^+ r$.
- We know that the reduction is confluent and that q reduces to 0. It implies that r reduces to 0. So we get $p \rightarrow_S^* 0$. □

We are now ready for the main step of the proof. In order to prove that the reduction is confluent, we show that it is sufficient that every spolynomial formed with polynomials of the basis reduces to 0:

$$(\forall p, q \in S,\ Spoly(p, q) \rightarrow_S^* 0) \Rightarrow \rightarrow_S^* \mathit{confluent}$$

We first prove two useful lemmas about the order defined in Section 2.1:
Lemma *StructLess:*

$$\forall t \in T_{A,M_n}, \forall p \in P_{A,M_n}, p <_p t \mathbin{\dot{+}} p$$

Lemma *RedLess:*

$$\forall S \in P_{A,M_n}\mathit{list}, \forall p, q \in P_{A,M_n}, p \rightarrow_S q \Rightarrow q <_p p$$

Note that the lemma *RedLess* and the fact that $<_p$ is well-founded ensure that the reduction always terminates. Now we can start the proof that the reduction is confluent:

- As the relation $<_p$ is well-founded, we prove that the reduction is confluent by induction on $<_p$ by taking as the main hypothesis that:

$$\forall p, q \in S,\ Spoly(p, q) \rightarrow_S^* 0$$

- Consider an arbitrary p, and suppose that

$$\forall q \in P_{A,M_n}, q <_p p \Rightarrow (\forall r, s \in P_{A,M_n}(q \rightarrow_S^* r \land q \rightarrow_S^* s) \Rightarrow r = s)$$

- We take two arbitrary reductions of p: $p \rightarrow_S^* r$ and $p \rightarrow_S^* s$ and prove that $r = s$.
- If p is irreducible, the property clearly holds $r = p = s$.
- Otherwise, consider p_1 and p_2 such that $p \rightarrow_S p_1 \rightarrow_S^* r$ and $p \rightarrow_S p_2 \rightarrow_S^* s$.
- Because $p_1 <_p p$ and $p_2 <_p p$, it is now sufficient to prove that there exists a p_3 such that $p_1 \rightarrow_S^* p_3$ and $p_2 \rightarrow_S^* p_3$ to get $r = p_3 = s$ by induction hypothesis.

- We do a case analysis on the nature of the reductions $p \rightarrow_S p_1$ and $p \rightarrow_S p_2$. There are four possible cases:
  1. Suppose $p = t \dotplus q \rightarrow_S t \dotplus q_1 = p_1$ and $p = t \dotplus q \rightarrow_S t \dotplus q_2 = p_2$.
     - Since $q <_p p$, $q \rightarrow_S q_1$, and $q \rightarrow_S q_2$, we get $reducef(S, q_1) = reducef(S, q) = reducef(S, q_2)$ by induction hypothesis.
     - It follows that $p_1 \rightarrow_S^+ t \dotplus reducef(S, q)$ and $p_2 \rightarrow_S^+ t \dotplus reducef(S, q)$.
     - It is then sufficient to take $p_3 = reducef(S, t \dotplus reducef(S, q))$.
  2. Suppose $p \rightarrow_S p /_p q_1 = p_1$ and $p = t \dotplus q \rightarrow_S t \dotplus q_2 = p_2$.
     - By definition of the one step division, there exists a polynomial q_3 such that $p /_p q_1 = q - q_3$.
     - Since $q \rightarrow_S q_2$, by applying the lemma *RedDistMinus*, there exists a polynomial q_4 such that $p_1 = q - q_3 \rightarrow_S^+ q_4$ and $q_2 - q_3 \rightarrow_S^+ q_4$.
     - It is easy to check that $q_2 - q_3 = p_2 /_p q_1$, so $p_2 \rightarrow_S^+ q_4$.
     - It is then sufficient to take $p_3 = reducef(S, q_4)$.
  3. Suppose $p = t \dotplus q \rightarrow_S t \dotplus q_1 = p_1$ and $p \rightarrow_S q /_p q_2 = p_2$.
     - This case is just the symmetric of case 2, so the property holds.
  4. Suppose $p \rightarrow_S p /_p q_1 = p_1$ and $p \rightarrow_S p /_p q_2 = p_2$.
     - p, q_1, and q_2 are non-zero polynomials, so $p = am \dotplus p'$, $q_1 = a_1 m_1 \dotplus q_1'$, and $q_2 = a_2 m_2 \dotplus q_2'$ for some $a, a_1, a_2 \in A$, some $m, m_1, m_2 \in M_n$, and some $p', q_1', q_2' \in P_{A,M_n}$.
     - q_1 and q_2 divide p, so m_1 and m_2 divide m. We deduce that there exists m_3 such that $m = m_3.(m_2 \hat{} m_1)$.
     - Using the definition of the one step division, we get that $p_1 - p_2 = (p' - (a /_a a_1)(m /_{M_n} m_1).q_1') - (p' - (a /_a a_2)(m /_{M_n} m_2).q_2')$
     - By simplifying the previous expression with the spolynomials definition we get $p_1 - p_2 = (am_3).Spoly(q_2, q_1)$.
     - Using the main hypothesis, we have $Spoly(q_2, q_1) \rightarrow_S^* 0$, so we get $p_1 - p_2 \rightarrow_S^* 0$.
     - By applying the lemma $Red^{+}Minus0$, there exists a polynomial p_4 such that $p_1 \rightarrow_S^+ p_4$ and $p_2 \rightarrow_S^+ p_4$.
     - It is then sufficient to take $p_3 = reducef(S, p_4)$.
- In all four cases, we are able to find such a polynomial p_3, so the property holds. □

Now in order to prove the theorem *BuchReduce*, it is sufficient to show

$$\forall p, q \in buch(S), Spoly(p, q) \rightarrow_{buch(S)}^* 0$$

This property is not immediate because the function *SpolyProd* does not generate all the possible spolynomials but only a reduced set. The following two lemmas:
Lemma *SpolyId:*

$$\forall p \in P_{A,M_n}, Spoly(p, p) = 0$$

Lemma *SpolySym:*

$$\forall p, q \in P_{A,M_n}, Spoly(p, q) = -Spoly(q, p)$$

ensure that the reduction to 0 of the reduced set implies the reduction of the complete set. This ends the proof of correctness.

4 The Proof of Termination

Except for the function *buchf*, all the proofs of termination of the functions we have been using are trivial: the arguments in recursive calls are always structurally smaller than the initial arguments.

For the termination of the function *buchf* we need a weak version of Dixon's lemma. This lemma states that in every infinite sequence of monomials M_n there exists at least one monomial M_i that divides another monomial M_j such that $i < j$. It follows that if we define the relation $\Re$ over list of polynomials as the smallest relation such that:

$$\begin{aligned}&\forall S \in P_{A,M_n}\, \mathit{list},\ \forall p \in P_{A,M_n},\\ &\qquad p \text{ is irreducible by } \rightarrow_S \wedge p \neq 0 \Rightarrow [p|S]\, \Re\, S\end{aligned}$$

the relation $\Re$ is well-founded. If now we define the relation $\Re'$ as the smallest relation such that:

- $\forall S, S', T, T' \in P_{A,M_n}\, \mathit{list},\ S\, \Re\, S' \Rightarrow (S,T)\, \Re'\, (S',T')$
- $\forall S, T \in P_{A,M_n}\, \mathit{list},\ \forall p \in P_{A,M_n},\ (S,T)\, \Re'\, (S,[p|T])$

$\Re'$ is the lexicographic product of two well-founded relations, so it is well-founded. Then for every recursive call within *buchf* it is easy to show that the argument y of the recursive call and the initial argument x are such that $y\, \Re'\, x$. So the function terminates.

5 Formalizing the Proofs Inside a Prover

One of the most satisfying aspect of our work has been to realize how naturally definitions and properties can be expressed in a higher order logic setting. What has been presented in Sections 2, 3 and 4 follows closely the proof development we have done in Coq. However we have avoided to present elements that were too specific to Coq. So we believe that the same definitions and the same proof steps could be used to get the proofs of correctness and termination in any theorem prover like Nuprl [4], HOL [8], Isabelle [18] or PVS [21], that allows the definition of recursive functions. In that respect we hope that what has been presented in the previous sections is a good compromise between the need for the proof to be human readable and the necessary detailed formalization due to mechanical theorem proving. In any case, it is a useful and important exercise to go from a textbook proof like the one in [7] to a proof that is suitable to mechanical theorem proving.

5.1 The Proof Development

The development in Coq is structured in three main parts:

1. The development of generic polynomials is composed of five modules. The module `porder` defines the notions of polynomials as lists of terms where terms are axiomatized and of ordered polynomials using an arbitrary order. Then the modules `seq`, `splus`, `smultm_lm`, and `sminus` define respectively equality, addition, term multiplication, and subtraction over polynomials.
2. The development of the algorithm itself contains five modules. The first two modules `spminus` and `sreduce` define respectively the one step division and the different notions of reduction. The module `def_spoly` defines the notion of spolynomials and proves that the reduction is confluent if all the spolynomials reduce to zero. The module `NBuch` defines an abstract version of the algorithm proving all the results with the help of some hypotheses. Finally, the module `Buch` instantiates the result of `NBuch` proving the different hypotheses.
3. The final part of the development is the instantiation. It is composed by three modules. The module `Monomials` defines monomials. The module `pair` defines terms as pairs of coefficients and monomials. The module `instan` glues all the different modules with the instantiation.

Figure 1 gives some quantitative information on the development. The columns correspond respectively to the number of lines of the module, the number of definitions, the number of theorems, the number of lemmas, and finally the ratio between the number of lines and the different objects defined or proved. Note that these figures do not include two important contributions that we have been using in the proof. A theory of lexicographic exponentiation derived from [17] is provided within the Coq system. It contains the main result needed for proving that reductions always terminate. A contribution by Loïc Pottier [19] gave us a non-constructive proof of the Dixon's lemma[1]. As explained before, this gives us indirectly the termination of the algorithm.

The proof development is around 9000 lines, so it represents an important effort. The proof has been carried out over a period of one year as a part-time activity. When we started, we thought the proof could be carried out in three months. Our first mistake was to underestimate the amount of work needed to formalize polynomials and the usual operations. The second lesson we have learned is that a special care has to be given to the organization of the development. Having a good set of definitions and basic properties is crucial when doing proofs. It is very often necessary to reorganize and reformulate definitions and theorems to increase reusability and productivity.

The other problems we have encountered are more specific to Coq. The entire proof development has been done using an arbitrary ring of coefficients and an arbitrary order. So each theorem of the development is fully quantified in order to allow later instantiation. But when we need to get theorems from a module

[1] It is the only non-constructive part of our proof.

Module	Lines	Definitions	Theorems	Lemmas	Ratio
porder	359	8	18	15	8
seq	359	8	17	8	10
splus	726	5	37	1	16
smultm_lm	201	1	19	2	9
sminus	500	4	25	2	16
Total$_1$	2145	26	116	28	12
spminus	380	2	19	0	18
sreduce	1439	16	34	9	24
def_spoly	1135	10	45	3	19
NBuch	455	13	26	0	11
Buch	1334	15	68	0	16
Total$_2$	4743	56	192	12	18
Monomials	408	12	18	4	12
pair	701	11	72	0	8
instan	943	18	11	0	32
Total$_3$	2052	41	101	4	14
Total	8940	123	409	44	15

Fig. 1. Quantitative information on the development

for a given instantiation, we need to operate individually on each of them which is very tedious. This is a well-known problem of modularity for which solutions have been proposed and implemented in other provers (see for example [6]). Clearly modularity is a must if we aim at large proof developments.

The equality we use for polynomials is not the simple structural equality. The polynomials we have defined may contain zero terms but we want to consider as equals those polynomials that only differ for zero terms. Also we want to take into account a possible equality $=_a$ over the elements of A. Using an explicit equality makes proofs harder in Coq because we miss the possibility to replace equals by equals. In order to regain substitutivity, we need to prove a theorem of compatibility for each function and predicate. For example, if $=_p$ denotes our equality over polynomials, it is necessary to prove the theorem:

$$p_1 =_p q_1 \wedge p_2 =_p q_2 \Rightarrow p_1 + p_2 =_p q_1 + q_2$$

to be allowed to replace polynomials in additions. Then proofs often get polluted with tedious steps of manipulation of the equality. In mathematics, the usual trick for avoiding this problem is to implicitly work with quotients. A real benefit could be gained in adding such a capability to Coq.

Finally if we look at Figure 1, the average of 15 lines per definition or theorem shows that proofs are often reasonably short. As a matter of fact, we have made very little use of automation. We mostly use the tactic `Auto` that simply takes a database of theorems and checks if the goal is a simple consequence of the database and the assumptions using the modus ponens only. It is difficult to evaluate what would be gained if we were doing the proof in a prover that provides more automation. Nevertheless, a specific class of goals we have encountered could largely benefit from automation. In the proof development, we construct the type of polynomials as being {`p:term list | (olist p)`}, i.e. the lists of terms such that the lists are ordered. In a proof that manipulates polynomials, it is often the case that we get several subgoals which require to prove that a list is ordered so it can be considered as a polynomial. Proving such subgoals is trivial most of the time but having to repeatedly prove them becomes quickly annoying.

5.2 Extracting the Algorithm

Once the development is finished, not only we have the proof of correctness of the algorithm but it is also possible to automatically extract an implementation. The self-contained version of the algorithm gives a 600 line long Ocaml program. The example below uses an instantiation of the algorithm with n variables polynomials for $n = 6$ over $\mathbb{Q}$ and the usual lexicographic order (`a` > `b` > `c` > `d` > `e` > `f`). Instantiating the implementation gives us 5 functions:

1. `gen: int -> poly` creates the generators;
2. `scal: int -> poly -> poly` multiplies the polynomial by an integer;
3. `plus: poly -> poly -> poly` adds two polynomials;
4. `mult: poly -> poly -> poly` multiplies two polynomials;
5. `buch: poly list -> poly list` computes the Gröbner basis.

We also write a prettyprinter in Ocaml to make the outputs of computation more readable. In the following, we present an interactive session with the toplevel Ocaml. Command lines are prefixed with # and end with two semicolons. We first define local variables to represent generators:

```
# let a = gen 0;;
val a : poly = a
# let b = gen 1;;
val b : poly = b
# let c = gen 2;;
val c : poly = c
# let d = gen 3;;
val d : poly = d
# let p1 =  gen 6;;
val p1 : poly = 1
```

We then construct the four n-cyclic polynomials for n=4:

```
# let r0 = (plus a (plus b (plus c d)));;
val r0 : poly = a +b +c +d
# let r1 = (plus (mult a b)
              (plus (mult b c) (plus (mult c d) (mult d a))));;
val r1 : poly = ab +ad +bc +cd
```

```
#let r2 = (plus (mult a (mult b c)) (plus (mult b (mult c d))
          (plus (mult c (mult d a)) (mult d (mult a b)))));;
val r2 : poly = abc +abd +acd +bcd
#let r3 = (plus (mult a (mult b (mult c d))) (scal (-1) p1));;
val r3 : poly = abcd -1
```

and the computation of the Gröbner basis gives:

```
# buch [r3;r2;r1;r0];;
- : poly list = [abcd -1; abc +abd +acd +bcd; ab +ad +bc +cd;
                 a +b +c +d; -b^2d -2bd^2 -d^3;  b^2 +2bd +d^2;
                 bcd^2 -bd^3 +c^2d^2 +cd^3 -d^4 -1;
                 bc -bd +c^2d^4 +cd -2d^2;
                 -bd^4 +b -d^5 +d; c^3d^3 +c^2d^4 -cd -d^2;
                 c^3d^2 +c^2d^3 -c -d; c^2d^6 -c^2d^2 -d^4 +1]
```

While the answer of the system for $n = 4$ was immediate, the computation for $n = 5$ had to be aborted after one hour of computation and a process size of more than 100Mb! This is not too surprising: the version of the algorithm is clearly too naive to perform well on large examples.

6 Related Work

Analytica [3] and more recently Theorema [22] propose an extension of the computer algebra system Mathematica [23] with a proving component. The examples they present are promising but their proof engines seem to need further developments in order to handle proofs of the same complexity as the one we have presented here. Also, there have been attempts to develop large fragments of mathematics within theorem provers. One of the first attempt was the Automath project [15]. The current largest attempt is the Mizar project [20]. Some recent efforts include Jackson's work on computational algebra [13] and Harrison's work on real analysis [9]. The focus of these works is mostly on formalizing mathematics inside a prover, so they give very few account of algorithmic aspects. Finally there have been several proposals to exploit a physical link between a prover and a computer algebra system to perform computation (see for example [1]). In [10], there is a discussion of some of the limitations of this approach.

As for the technique of program extraction, it has been demonstrated mostly on toy programs [11], [16]. We believe that our algorithm is one of the first non-trivial examples using this technique.

7 Conclusion and Future Work

While working on this development we had clearly the feeling to be at the frontier between proving and computing. Even if we were mostly in the proving world trying to state properties about polynomials, we were also able to test and compute with these very same polynomials. The situation is not yet ideal and we have described some of the problems we have encountered. Still we hope that this experiment shows that we are not so far from being able to mix proving and computing.

It is interesting to contrast the 9000 lines of the proof development with the 600 lines of the extracted Ocaml implementation of the algorithm. Proving requires much more effort than programming. This is not a surprise. It also indicates that the perspective of developing a completely certified computer algebra system is unrealistic for the moment. The first step in that direction is definitely to increase the knowledge of provers with basic algebraic notions. One third of our proof lines has been used to construct a library of multivariate polynomials. More automation and a better support to structure the development are also mandatory.

The work we have done on Buchberger's algorithm is far from being finished. Our algorithm is a textbook version of a real algorithm. We are aware that we still need to give evidence that with our approach we can obtain an algorithm that can be compared with what is proposed in general-purpose computer algebra systems. In that respect, it is worth noticing that correctness becomes an important issue for optimized versions of the algorithm. The main optimization consists in avoiding to check the reducibility to zero of some spolynomials. A common implementation error is to be too aggressive in the optimization and discard spolynomials that are in fact not reducing to zero. Even in that case, the algorithm can still behave well because the generation of spolynomials is heavily redundant. Testing may not be sufficient to spot this kind of implementation error.

Moreover, we would like to investigate the possibility of obtaining automatically or semi-automatically a textbook version of the proof of correctness of the algorithm directly from our development. In [5], a method is proposed to automatically produce a document in a pseudo-natural language out of proofs in Coq. Applying this method to our complete development seems very promising.

There are several ways in which this initial experiment can be extended. First of all it would be very interesting to see how the same proof looks like in other theorem proving systems. It would give a more accurate view of what current theorem proving technology can achieve on this particular problem. Also, we plan to complement this initial contribution with the certification of other standard algorithms for polynomials such as factorization. Our long term goal is to provide a completely certified kernel for non-trivial polynomial manipulations.

References

1. Jacques Calmet and Karsten Homann. Classification of communication and cooperation mechanisms for logical and symbolic computation systems. In *First International Workshop 'Frontiers of Combining Systems' (FroCoS'96)*, Kluwer Series on Applied Logic, pages 133–146. Springer-Verlag, 1996.
2. Bruce W. Char, Keith O. Geddes, and Gaston H. Gonnet. *First leaves: a tutorial introduction to Maple V.* Springer-Verlag, 1992.

3. Edmund Clarke and Xudong Zhao. Analytica — a theorem prover for Mathematica. Research report, Carnegie Mellon University, 1991.
4. Robert L. Constable, Stuart F. Allen, H.M. Bromley, Walter R. Cleaveland, James F. Cremer, Robert W. Harper, Douglas J. Howe, Todd B. Knoblock, Nax P. Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing mathematics with the Nuprl proof development system.* Prentice Hall, 1986.
5. Yann Coscoy, Gilles Kahn, and Laurent Théry. Extracting text from proofs. In *Typed Lambda Calculus and its Applications*, volume 902 of *LNCS*, pages 109–123. Springer-Verlag, 1995.
6. William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. Little theories. In D. Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *LNCS*, pages 567–581. Springer-Verlag, 1992.
7. Keith O. Geddes, Stephen R. Czapor, and George Labahn. *Algorithms for computer algebra.* Kluwer, 1992.
8. Michael Gordon and Thomas Melham. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, 1993.
9. John R. Harrison. Theorem proving with the real numbers. Technical Report 408, University of Cambridge Computer Laboratory, 1996. PhD thesis.
10. John R. Harrison and Laurent Théry. Extending the HOL theorem prover with a computer algebra system to reason about the reals. In *Higher Order Logic Theorem Proving and Its Applications*, volume 780 of *LNCS*. Springer-Verlag, August 1995.
11. Doug Howe. Reasoning About Functional Programs in Nuprl. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *LNCS*, pages 144–164. Springer-Verlag, 1993.
12. Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The Coq proof assistant: A tutorial: Version 6.1. Technical Report 204, INRIA, 1997.
13. Paul B. Jackson. Enhancing the Nuprl proof development system and applying it to computational abstract algebra. Technical Report TR95-1509, Cornell University, 1995.
14. Xavier Leroy. Objective Caml. Available at `http://pauillac.inria.fr/ocaml/`, 1997.
15. Rob P. Nederpelt, J. Herman Geuvers, and Roel C. De Vrijer, editors. *Selected papers on Automath.* North-Holland, 1994.
16. Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15(5-6):607–640, May–June 1993.
17. Lawrence C. Paulson. Constructing Recursion Operators in Intuitionistic Type Theory. *Journal of Symbolic Computation*, 2(4):325–355, December 1986.
18. Lawrence C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *LNCS*. Springer-Verlag, 1994.
19. Loïc Pottier. Dixon's lemma. Available at `ftp://www.inria.fr/safir/pottier/MON/`, 1996.
20. Piotr Rudnicki. An overview of the MIZAR projet. In *Workshop on Types and Proofs for Programs.* Available by ftp at `pub/csreports/Bastad92/proc.ps.Z` on `ftp.cs.chalmers.se`, June 1992.
21. John M. Rushby, Natajaran Shankar, and Mandayam Srivas. PVS: Combining specification, proof checking, and model checking. In *CAV '96*, volume 1102 of *LNCS*. Springer-Verlag, July 1996.

22. Daniela Vasaru, Tudor Jebelean, and Bruno Buchberger. Theorema: A system for formal scientific training in natural language presentation. Technical Report 97-34, Risc-Linz, 1997.
23. Stephen Wolfram. *Mathematica: a system for doing mathematics by computer.* Addison-Wesley, 1988.