INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Proving and Computing: a certified version of the Buchberger's algorithm

Laurent Théry

N° 3275

October 1997

THÈME 2

Rapport de recherche

ISSN 0249-6399

Proving and Computing: a certified version of the Buchberger's algorithm

Laurent Théry

Thème 2 — Génie logiciel
et calcul symbolique
Projet CROAP

Rapport de recherche n° 3275 — October 1997 — 27 pages

Abstract: This paper shows on a non-trivial example that it is possible to mix proving and computing using current technologies. We present a proof of the 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.

Key-words: Buchberger's algorithm, theorem proving, computer algebra, Coq

(Résumé : tsvp)

Unité de recherche INRIA Sophia Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA ANTIPOLIS Cedex (France)
Téléphone : 04 93 65 77 77 – international : +33 4 93 65 77 77 – Fax : 04 93 65 77 65 – international : +33 4 93 65 77 65

Preuve et Calcul Formel: une version certifiée de l'algorithme de Buchberger

Résumé : Ce papier montre sur un exemple non trivial qu'il est possible, avec les technologies actuelles, de conjuguer preuve et calcul formel. Nous présentons une preuve de l'algorithme de Buchberger développée sous l'assistant de preuve Coq. L'algorithme defini dans Coq peut alors être efficacement compilé et utilisé pour le calcul formel.

Mots-clé : algorithme de Buchberger, démonstration automatique, calcul formel, Coq

1 Introduction

The community that is doing mathematics on computer is split in two. On the one hand, there are people that are mainly interested in computing and solving. They have developed very efficient tools, *computer algebra systems*, to perform their computations. Computer algebra systems can either be general-purpose such as Axiom[14], Maple[3] and Mathematica[23] or dedicated to specific areas such as Gb[8] and Macaulay[1], just to cite systems connected with Gröbner basis. These systems have been really successful. They are used on daily basis by applied mathematicians and engineers. They are also used heavily for educational purposes. On the other hand, there are people that are mainly interested in the proving activity. They have also developed tools, *theorem provers*, to mechanize proofs on computer. Theorem provers range from fully automatic systems such as Nqthm[2] and Otter[24], to mostly interactive ones such as Coq[12], HOL[10], Mizar[20], and Nuprl[5], through systems that try to find a good compromise between automation and interactivity such as Isabelle[18] and PVS[21]. In our opinion, theorem provers have had less impact than computer algebra systems. The main reason is that, while computer algebra systems have quickly surpassed the computation power of human beings, theorem provers are still (and perhaps will always be) far from proving what any mathematician can easily prove. Nevertheless, there have been attempts to develop large fragments of mathematics within theorem provers. One of the first attempt has been the Automath project [16]. Some recent efforts include Jackson's work on computational algebra [13], Harrison's work on real analysis [11], and Shankar's work on Gödel's theorems [22]. Finally the largest current attempt is, without any doubt, the Mizar project [20].

There would be obvious benefits in having a framework where both proving and computing are possible:

- Algorithms in computer algebra systems are usually equiped with little knowledge of their applicability and correctness. 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. This problem is even more crucial for general-purpose computer algebra systems where the library of algorithms is mostly developed by the user community. It is a bit deceiving that extensions of the system can be performed without giving some evidence of their correctness. Adding a proving component to computer algebra systems would make it possible to state and prove properties about algorithms.
- Theorem provers usually come with very little computation power. This makes it difficult to complete proofs for which some computation steps are needed. Even simple mathematical facts such that the primality of small numbers are often hard to establish inside a prover.
- Finally for educational purposes, it would be ideal to have a system where one could define mathematical objects and then both compute and prove properties about them.

Building such a system from scratch would require an important effort. A more pragmatic approach consists in complementing an existent system. 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 language used by Maple has no notion of local variables inside a procedure. Thus, stating properties of algorithms tunrs 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. Writing efficient compilers inside a prover is still an open question. An alternative solution is for the system to be able to translate its algorithms in another programming language that has a compiler.

Our approach follows the second line. We have chosen the theorem prover Coq to do our experiments. Coq appears an appropriate choice. First of all, it is a prover based on type theory, so it manipulates objects with a rich notion of types. The Axiom system has shown the suitability of types to describe mathematical objects in computer algebra. Second, Coq proposes an extraction mechanism that, given an algorithm defined in the system, generates an implementation in the language Ocaml [15] 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 to these questions that we decided to work on the proof of correctness of the Buchberger's algorithm. We started from a five page description of the algorithm in a standard introduction book [9]. The goal was simple: to develop enough mathematical knowledge in Coq for stating the algorithm and proving its correctness.

The paper is organized as follows. In Section 2 we introduce the Buchberger's algorithm. In Second 3 we sketch its proof of correctness. In Section 4 we present Coq and explain the main steps of our development. Finally we draw some conclusions and discuss future work.

2 The Buchberger's algorithm

The 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 [9].

2.1 Polynomials and usual operations

In the following we consider usual polynomials of dimension n over an arbitrary field $(A,+_a,-_a,*_a,/_a,0_a,1_a)$. When illustrating definitions with examples, we are usually considering polynomials of low dimension ($n = 2$ or $n = 3$) over the real numbers ($A = \mathbb{R}$).

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 monomial is denoted by M_n where n is the dimension. The set of terms and polynomials are denoted by T_{A,M_n} and P_{A,M_n} respectively. To give a concrete example, consider the following polynomial '$2y^3 + 5xy^4 - 3$'. Its first term is composed of the coefficient '2' and the monomial 'y^3' while the last term is composed of the coefficient '-3' and the null monomial 'x^0y^0'. In the following we note the null monomial 1_{M_n} and 0 the null polynomial.

We consider two operations over these polynomials: addition and multiplication by a term. Adding two polynomials consists in merging the list of terms by adding the coefficients of terms with the same monomial. For example, adding the two polynomials '$2y^3 + 5xy^4 - 3$' and '$x^4 - 2xy^4 + x - 2$' gives the polynomial '$x^4 + 2y^3 + 3xy^4 + x - 5$'. Multiplying a polynomial by a term consists in multiplying each term of the polynomials by the term. We multiply two terms by simply multiplying pairwise their coefficients and monomials. So for example, multiplying the polynomial '$y^3 - xy^2 - 3$' and the given term '$2x^2$' gives the polynomial '$2x^2y^3 - 2x^3y^2 - 6x^2$'. In the following, we overload the symbol '.' to denote both multiplication of monomials and multiplication by a term.

2.2 Ordered polynomial

An *order* $<_{M_n}$ over monomials is a binary relation that is

- transitive: $\forall m_1, m_2, m_3 \in M_n, (m_1 <_{M_n} m_2 \wedge m_2 <_{M_n} m_3) \Rightarrow m_1 <_{M_n} m_3$
- irreflexive: $\forall m \in M_n, \neg(m <_{M_n} m)$.

An order $<_{M_n}$ over monomials is *total* if

$$\forall m_1, m_2 \in M_n, (m_1 <_{M_n} m_2) \vee (m_2 <_{M_n} m_1) \vee (m_1 = m_2)$$

An order $<_{M_n}$ over monomials is *well-founded* if there exists no infinite decreasing sequence of monomials:

$$(\forall k \in N, m_k \in M_n) \Rightarrow \exists k_1 \in N, (m_{k_1} = m_{k_1+1}) \vee (m_{k_1} <_{M_n} m_{k_1+1})$$

An order $<_{M_n}$ over monomials is said to be *admissible* if

- the null monomial is minimal for the order: $\forall m \in M_n,\ \neg(m <_{M_n} 1_{M_n})$
- the order is compatible with monomial multiplication:

$$\forall m, n, s \in M_n,\ m <_{M_n} n \Rightarrow s.m <_{M_n} s.n$$

A standard order is the *lexicographic* order. We first choose an arbitrary ordering $<_v$ on the variables representing the dimension $x_n <_v x_{n-1} \ldots <_v x_2 <_v x_1$. Then, we define the lexicographic order $<_L$ as:

$$\forall m_1, m_2 \in M_n, m_1 = x_1^{i_1} x_2^{i_2} \ldots x_n^{i_n} <_L x_1^{j_1} x_2^{j_2} \ldots x_n^{j_n} = m_2$$

$$\Longleftrightarrow$$

$$\exists l \leq n, i_l < j_l \wedge (\forall k < l,\ i_k = j_k)$$

It is easy to check that this relation is a total, well-founded and admissible order.

Given an admissible well-founded total order $<_{M_n}$ over monomials, it is possible to represent a polynomial as an ordered list of terms, such that the list of corresponding monomials is ordered, i.e. each monomial in the list is greater than the ones at its right. For example, the polynomial '$2x^2y + 2xy^4 + 5$' is ordered with respect to the lexicographic order where $y <_v x$. In order to stress the ordered representation of the polynomial, we use the symbol $\stackrel{>}{+}$. So, when needed, the previous polynomial will be written '$2x^2y \stackrel{>}{+} 2xy^4 \stackrel{>}{+} 5$'. In the following, $<_{M_n}$ always denotes an admissible well-founded total order over monomials.

2.3 Normal form

Given the definition of polynomials, it is possible that polynomials carry terms with null coefficient. Equality over 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 can first define the function *nf* that computes the normal form of a polynomial by removing terms with null coefficient:

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

It is then possible to state that:

$$\forall p, q \in P_{A,M_n},\ p = q \iff nf(p) =_s nf(q)$$

where the equality on the right is the simple structural equality.

2.4 One step division, reduction and irreducibility

We first define division over monomials. Since the division 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:

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

This division can be lifted to polynomials, so we get the one step division $/_p$ defined as follows :

$$\begin{array}{l}\forall m_1, m_2 \in M_n,\ divP_{M_n}(m_1, m_2) \Rightarrow \\ \qquad \forall a_1, a_2 \in A, \forall p_1, p_2 \in P_{A,M_n},\ a_2 \neq 0_a \Rightarrow\end{array}$$

$$(a_1 m_1 \stackrel{>}{+} p_1) /_p (a_2 m_2 \stackrel{>}{+} p_2) = p_1 - (a_1 /_a a_2)(m_1 /_{M_n} m_2).p_2$$

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

- $\forall p_1, p_2 \in P_{A,M_n}, \forall t \in T_{A,M_n}, p_1 \rightarrow_S p_2 \Rightarrow t \stackrel{>}{+} p_1 \rightarrow_S t \stackrel{>}{+} p_2$
- $\forall m_1, m_2 \in M_n,\ divP_{M_n}(m_1, m_2) \Rightarrow$

  $\forall a_1, a_2 \in A, \forall p_1, p_2 \in P_{A,M_n},\ a_2 \neq 0_a \Rightarrow$

  $$(a_2 m_2 \stackrel{>}{+} p_2) \in S \Rightarrow a_1 m_1 \stackrel{>}{+} p_1 \rightarrow_S (a_1 m_1 \stackrel{>}{+} p_1) /_p (a_2 m_2 \stackrel{>}{+} p_2)$$

We can now define the reflexive transitive closure of the relation $\rightarrow_S$. The relation $\rightarrow_S^+$ is defined as the smallest relation such that:

- $\forall p \in P_{A,M_n},\ p \rightarrow_S^+ p$
- $\forall p_1, p_2, p_3 \in P_{A,M_n},\ p_1 \rightarrow_S p_2 \Rightarrow p_2 \rightarrow_S^+ p_3 \Rightarrow p_1 \rightarrow_S^+ p_3$

We can also define the predicate of irreducibility $irreducible_S$ as follows:

$$irreducible_S(p) \iff \forall q \in P_{A,M_n}, \neg(p \rightarrow_S q)$$

This leads to the reduction $\rightarrow_S^*$ defined as follows:

$$p \rightarrow_S^* q \iff p \rightarrow_S^+ q \wedge irreducible_S(q)$$

2.5 Spolynomials

Given two monomials, there exists a least common multiplier. For this, we simply take the maximal exponent in each dimension. For example, we have $lcm(x^2y, xy^2z^3) = x^2y^2z^3$. We define the function $Spoly$ as follows

- $\forall p \in P_{A,M_n},\ Spoly(p, 0) = 0$
- $\forall p \in P_{A,M_n},\ Spoly(0, p) = 0$
- $\forall m \in M_n\, \forall p, q \in P_{A,M_n},\ Spoly(p, 0_a m \stackrel{>}{+} q) = Spoly(p, q)$
- $\forall m \in M_n, \forall p, q \in P_{A,M_n},\ Spoly(0_a m \stackrel{>}{+} p, q) = Spoly(p, q)$
- $\forall m_1, m_2 \in M_n, \forall a_1, a_2 \in A, \forall p_1, p_2 \in P_{A,M_n},$
  $((p = a_1 m_1 \stackrel{>}{+} p_1) \wedge (q = a_2 m_2 \stackrel{>}{+} p_2) \wedge a_1 \neq 0_a \wedge a_2 \neq 0_a) \Rightarrow$
  $Spoly(p, q) = ((1_a /_a a_1) lcm(m_1, m_2) /_{M_n} m_1).p_1 - ((1_a /_a a_2) lcm(m_1, m_2) /_{M_n} m_2).p_2$

In the last case, the polynomial $lcm(m_1, m_2)$ represents the smallest polynomial that can be divided by both polynomials p and q:

$$lcm(m1, m2) \rightarrow_{\{p\}} q_1 = -(1_a /_a a_1)(lcm(m_1, m_2) /_{M_n} m_1).p_1$$

$$lcm(m1, m2) \rightarrow_{\{q\}} q_2 = -(1_a /_a a_2)(lcm(m_1, m_2) /_{M_n} m_2).p_2$$

The function *Spoly* corresponds to their difference: $Spoly(p, q) = q_2 - q_1$

2.6 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 of 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 $<S> = I$.

2.7 Gröbner basis and the 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 polynomial:

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

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

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

For the moment, assume that such a function exists. The algorithm is a completion that takes two arguments, namely the initial list and 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 candidate list is empty, the basis is returned (first case). If the head of the candidate list does not reduce to zero, it is added to the basis and the spolynomials computed by *SpolyL* are added to the completion list (second case). If the head of the candidate list reduces to zero, recursively calls are made with the tail of the candidate list (third case).

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 *BuchfStable:*

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

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

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

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 *BuchfReduce* 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 we 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 *BuchfReduce* 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 $<S>$ and want to prove that $p \rightarrow_S^* 0$.
- 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 theorem $Red^{+}Minus0$, 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 $p \rightarrow_S^* 0$. $\square$

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:

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

We first define an order $<_p$ over polynomials based on the order over monomials as the smallest relation such that:

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

This defines a total well-founded order. Given this order, we can prove two useful lemmas:
Lemma *StructLess:*

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

Lemma *RedLess:*

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

Note that the theorem *RedLess* and the fact that $<_p$ is well-founded ensure that the reduction always terminates. Now we have enough material to start the proof that the reduction is confluent:

- As the order $<_p$ 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

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

- 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 $p = t \stackrel{>}{+} q \rightarrow_S t \stackrel{>}{+} q_1 = p_1$ and $p = t \stackrel{>}{+} q \rightarrow_S t \stackrel{>}{+} q_2 = p_2$.
     - Since $q <_p p$, $q \rightarrow_S q_1$, and $q \rightarrow_S q_2$, we get $\mathit{reducef}(S, q_1) = \mathit{reducef}(S, q)$ $= \mathit{reducef}(S, q_2)$ by induction hypothesis.
     - It follows that $p_1 \rightarrow_S^+ t \stackrel{>}{+} \mathit{reducef}(S, q)$ and $p_2 \rightarrow_S^+ t \stackrel{>}{+} \mathit{reducef}(S, q)$.
     - It is then sufficient to take $p_3 = \mathit{reducef}(S, t \stackrel{>}{+} \mathit{reducef}(S, q))$.
  2. Suppose $p \rightarrow_S p/_p q_1 = p_1$ and $p = t \stackrel{>}{+} q \rightarrow_S t \stackrel{>}{+} q_2 = p_2$.
     - 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 theorem *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 $p_3 = \mathit{reducef}(S, q_4)$.
  3. Suppose $p = t \stackrel{>}{+} q \rightarrow_S t \stackrel{>}{+} q_1 = p_1$ 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 $p = am \stackrel{>}{+} p'$, $q_1 = a_1 m_1 \stackrel{>}{+} q_1'$, and $q_2 = a_2 m_2 \stackrel{>}{+} q_2'$ 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 $m = m_3.lcm(m_2, m_1)$.
- 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, we get:

$$p_1 - p_2 = (am_3)((1_a/_a a_2)(lcm(m_2, m_1)/_{M_n} m_2).q_2' - (1_a/_a a_1)(lcm(m_2, m_1)/_{M_n} m_1).q_1')$$

- By definition of the spolynomials, we get $p_1 - p_2 = (am_3).Spoly(q_2, q_1)$
- Using the main hypothesis, we have $Spoly(q_2, q_1) \to_S^* 0$, so we get $p1 - p2 \to_S^* 0$.
- By applying the theorem $Red^+Minus0$, there exists a polynomial p_4 such that $p_1 \to_S^+ p_4$ and $p_2 \to_S^+ p_4$.
- 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 *BuchfReduce*, it is sufficient to show

$$Q = buchf(nfL(S), SpolyProd(nfL(S))) \Rightarrow \forall p, q \in Q, Spoly(p, q) \to_Q^* 0$$

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 Formalizing the proof in a theorem prover

What has been presented in Section 2 and 3 follows 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 those proof steps could be used to get the proof of correctness in any theorem prover like Nuprl, HOL or PVS, 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 [9] to a proof that is suitable to mechanical theorem proving.

In this section, we first give an overview of the system Coq and show how some of the definitions in Section 2 can be entered into the system. Second, we provide some more quantitative information on the proof development, trying to summarize the difficulties we encountered in the formalization. Finally, we show some running examples of the extracted algorithm.

4.1 Coq

Rather than giving a complete introduction to Coq, which can be found for example in [12], we introduce the basic notions of Coq using the definition of the algorithm in Section 2.7. Coq is a prover based on type theory. The isomorphism of Curry-Howard shows the correspondence between propositions and types, proofs and terms. A proposition is then represented in Coq by a type and a proof is given by a term that has the given type. Coq uses the calculus of construction, a higher order lambda-calculus, to express formulae and proofs. Type theoretic theorem provers are constructive by nature. They see the proof of the proposition $A \Rightarrow B$ as a function that, given a proof of A, returns a proof of B. The calculus of construction has been extended to the calculus of inductive construction to handle inductive constructs. The user interacts with Coq with what is called the *vernacular* that allows the user to structure the development.

The first example we give is the definition of a polymorphic list. This definition is very similar to what would be done in a functional language à la ML:

```
Section PolyList.
Variable A: Set.

Inductive list : Set :=
       nil : list
    | cons : A -> list -> list.

End PolyList.
```

The first line defines a new section. The second line declares an arbitrary type `A`. Writing that this new type is of type `Set` simply indicates that this type represents objects by opposition to the other base type `Prop` that represents propositions. The next line is defining a new datatype `list` to represent lists of objects of type `A`. The definition is inductive and a list is itself an object. It has two constructors `nil` and `cons`. The first constructor represents the empty list while the second constructor takes an element and a list to form a list. This definition being inductive, inductive theorems are automatically generated. For example, the one for induction over lists for an arbitrary predicate P is:

```
Check list_ind.
list_ind
     : (P:list->Prop)
        (P nil)->((a:A)(l:list)(P l)->(P (cons a l)))->(l:list)(P l)
```

which printed more nicely gives:
Theorem *list_ind***:**

$$\forall P\!:\!\texttt{list} \rightarrow \texttt{Prop},\ (P\,\texttt{nil}) \Rightarrow (\forall a\!:\!A,\ \forall l\!:\!\texttt{list},\ (P\,l) \Rightarrow (P\,(\texttt{cons}\,a\,l))) \Rightarrow (\forall l\!:\!\texttt{list},\ (P\,l))$$

The last line of the definition of a polymorphic list closes the section. Closing the section generalizes all the constructors that use variables. Thus, after the section, `list` is of type `Set` $\rightarrow$ `Set` and `nil` and `cons` can now be parameterized by an arbitrary type `A`. Their types are respectively $\forall A : \texttt{Set}, (\texttt{list}\,A)$ and $\forall A : \texttt{Set}, A \rightarrow (\texttt{list}\,A) \rightarrow (\texttt{list}\,A)$.

In order to define the function *SpolyProd*, we assume we have already a type `poly` to represent polynomials and a function `spoly` to compute spolynomials:

```
Parameter poly: Set.
Parameter spoly: poly -> poly -> poly.
```

We can now define the first function *SpolyL*:

```
Fixpoint SpolyL[p:poly; L1, L2:(list poly)]: (list poly) :=
   Cases L2 of
         nil => L1
      | (cons q L2) => (cons poly (spoly p q) (SpolyL p L1 L2))
   end.
```

In Coq every function must terminate. The termination of the function *SpolyL* can be automatically deduced by the system because the last argument of the recursive call is a strict subterm of the initial last argument. Note also that only objects are matched when pattern matching terms (so we write '`cons q L2`'), while we need to add explicitly type information when constructing terms (so we write '`cons poly ..`'). A possible alternative provided by Coq is to replace types that can be inferred by the symbol '`?`'.

The function *SpolyProd* is defined in a very similar way. Termination of this function is also automatically deduced by a simple structural argument:

```
Fixpoint SpolyProd[L:(list poly)]: (list poly) :=
   Cases L of
         nil => (nil ?)
      | (cons p L) => (SpolyL p (SpolyProd L) L)
   end.
```

It is now possible to do some computation using the system. We define three arbitrary polynomials and ask to compute the associated list of spolynomials:

```
Parameters p1, p2, p3:poly.
Compute (SpolyProd (cons ? p1 (cons ? p2 (cons ? p3 (nil ?))))).
```

The answer of the system is the expected one:

```
= (cons poly (spoly p1 p2)
         (cons poly (spoly p1 p3) (cons poly (spoly p2 p3) (nil poly))))
     : (list poly)
```

In order to formalize the Buchberger's algorithm, we first need to define some extra parameters:

```
Parameter reducef:(list poly) -> poly -> poly.
Parameter zeroP: poly -> bool.
Parameter nf:poly ->poly.
```

A first attempt to define the Buchberger's algorithm is the following:

```
Fixpoint buchf [pair:(list poly) * (list poly)] : (list poly) :=
  let (l1, l2) = pair in
     Cases l2 of
       nil => l1
     | (cons p l2) =>
          if (zeroP (nf (reducef l1 p))) then
            (buchf (l1, l2))
                        else
            (buchf ((cons ? (nf (reducef l1 p)) l1),
                      (SpolyL (nf (reducef l1 p)) l2 l1)))
     end
  .
```

Unfortunately the termination of this function is far from being obvious. So the system refuses the definition. The termination of this function is given by a lexicographic order on pairs of lists of polynomials. The order on the left argument of the pair is deduced from 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 inserting in a list of monomials a monomial that it is not divisible by any of the other monomials is well-founded. This gives a well-founded order on lists of polynomials by just taking the leading monomials of each polynomials. The order on the right argument of the pair is the simple structural order on lists.

Let us suppose that we have already defined such a lexicographic order in Coq:

```
Parameter lexPair:
     (list poly) * (list poly) -> (list poly) * (list poly) -> Prop.
```

Since the termination of the function '`buchf`' cannot be deduced automatically by the system, we first tell the system that we are going to define a function of a given type:

```
Definition buchf: (list poly) * (list poly) ->(list poly).
```

With this line, we enter into the proof mode where we are challenged to show that there exists a function with such a type. We can then use the tactic `Realizer` to provide directly the witness function[1]:

```
Realizer  <(list poly)> rec buchfx :: :: {lexPair}
  [pair:(list poly) * (list poly)]
   let (l1, l2) = pair in
     Cases l2 of
       nil => l1
     | (cons p l2) =>
          if (zeroP (nf (reducef l1 p))) then
            (buchfx (l1, l2))
                        else
            (buchfx ((cons ? (nf (reducef l1 p)) l1),
                           (SpolyL (nf (reducef l1 p)) l2 l1)))
     end.
```

Here the witness function is the function *buchfx*. We explicitly give the order `lexPair` that proves termination. Once the witness has been given, we apply the tactic `Program_all` to construct step by step the function using the realizing function as a guideline. We are then left with three goals to prove:

```
3 subgoals

  a : (list poly)*(list poly)
  ============================
   (well_founded (list poly)*(list poly) lexPair)

subgoal 2 is:
 (lexPair (l1,l0) (l1,(cons poly r1 l0)))
subgoal 3 is:
 (lexPair
   ((cons poly (nf (reducef l1 r1)) l1),
    (SpolyL (nf (reducef l1 r1)) l0 l1)) (l1,(cons poly r1 l0)))
```

We first need to prove that the order `lexPair` is well-founded. This is straightforward because the order is a lexicographic composition of two well-founded orders. The second goal needs a proof that the argument of the recursive call of the `then` part is smaller than the initial argument with respect to the order. This is obvious because the second component of the first pair is structurally smaller than the one of the second pair. The third goal corresponds to the recursive call of the `else` part. The polynomials '`(nf (reducef l1`

[1]To make the code more readable, we did not write all the type information needed for this function to be accepted by Coq

p))' is not the null polynomial so its leading monomial is not divisible by any of the leading monomials of the polynomials in l1 since it is irreducible by l1.

Once the three goals have been proved, we have defined the function we wanted. Unfortunately this is not enough to be able to reason on this function. With the current version of Coq it is very difficult to do symbolic computation with functions defined with an arbitrary order. The usual technique to overcome this problem is first to define a predicate that represents the function. Predicates are more easy to manipulate than functions. The predicate version of the previous function buchf is the following:

```
Inductive BuchP: ((list poly) * (list poly)) -> (list poly) ->Prop :=
    BuchP0: (L1:(list poly))(BuchP (L1, (nil poly)) L1)
  | BuchP1:
      (a:poly) (L, L1, L2: (list poly))
       (zeroP (nf (reducef L1 a))) = false ->
       (BuchP
         ((cons poly (nf (reducef L1 a)) L1),
          (SpolyL (nf (reducef L1 a)) L1 L2))
         L)->
      (BuchP (L1, (cons poly a L2)) L)
  | BuchP2:
      (a:poly) (L, L1, L2:(list poly))
      (zeroP (nf (reducef L1 a))) = true -> (BuchP (L1,L2) L) ->
          (BuchP (L1,(cons poly a L2)) L).
```

Second, we define the function buch with a refined type:

```
Definition buchf:
  (pair:(list poly) * (list poly)) -> {q:(list poly) | (BuchP pair q)}.
```

The last type should be read as the set of polynomials q such that 'BuchP pair q' holds. So we impose that the result must verify the predicate.

Now applying the same realizer plus the tactic Program_all generates six goals: the previous three ones, plus three new ones to prove that the result of the function verifies the predicate. These three new goals are directly solved by the application of one of the three constructors BuchP0, BuchP1, and BuchP2. The advantage of having the refined type for the function is that we get for free the following theorem:

$$\forall L_1, L_2 : (\texttt{list poly}), (\texttt{BuchP}\,(L_1, L_2)\,(\texttt{buchf}\,(L_1, L_2)))$$

Thus, every property proved for the predicate can be lifted to the function.

4.2 The proof development

The previous section has shown that with little effort it is possible to enter algorithms in Coq. The main constraint imposed by the system is that every algorithm should come along with a proof of its termination. However, those only interested in the computational aspect of the algorithm can always skip this phase and state the termination as an axiom. Even when using axioms, it is still possible to extract algorithms, compile them and execute them in Ocaml.

The benefit of having defined the algorithm in Coq is that it is now possible to state and prove properties about the algorithm. Note that the simple fact of having entered the algorithm into a typed language means that we have already proved some of its properties. For example, if `poly` represents the type of ordered polynomials, defining the addition of polynomials as a function of type `poly -> poly -> poly` already shows that adding two ordered polynomials will always give a polynomial that is ordered. So data are never corrupted.

Two properties of the Buchberger's algorithm are important: termination and correctness. The former states that the algorithm always terminates. The later states that, whenever the algorithm terminates, it returns a Gröbner basis. The proofs of termination and correctness can be done at any arbitrary level of detail. We may either axiomatize a certain number of basic properties and show that termination and correctness logically follow from them or be more ambitious and want to prove these theorems from the very definitions. The second approach is intellectually the most satisfying but also the most involving. As we were experimenting with proving properties of computer algebra algorithms, we have chosen the second approach mainly to evaluate the amount of work the complete development would represent.

We started the development by defining polynomials. In a first approximation, we decided to skip the definition of monomials and terms. So, we just axiomatized terms. Our main concern was to make sure that the proofs on polynomials would be properly abstracted from terms, i.e no information about the concrete representation of terms should be used in the proofs about polynomials. The development has been structures 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 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 prove 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.

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

Figure 1: Quantitative information on the development

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 of Loïc Pottier [19] gave us a non-constructive proof of the Dixon's lemma[2]. 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. We estimate that it represents a six-month full time effort. 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. With a bad set of basic properties it is always possible to prove the new properties we want to prove, but the proofs end up to be unnatural and unnecessarly intricate. This implies that

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

the development does not grow linearly, while it is very often necessary to reorganize and reformulate definitions and theorems to increase reusability and productivity. The other problems that make the proof more difficult than expected are more technical and mostly specific to Coq. Below we recall the three most important ones that deal with *modularity*, *equality* and *automation*.

The entire proof development uses an arbitrary ring of coefficients. Thus, it is necessary to abstract the ring inside the development while leaving the possibility to instantiate the different results to a particular ring outside the development. This is a well-known problem of modularity for which solutions have been proposed and implemented in other provers (see for example [7]). Coq proposes such a mecanism for abstraction and instantiation but only at the level of a section. Hypotheses inside a section are automatically generalized outside the section. Unfortunately it is impossible to keep the development inside a single section. First of all, the development is naturally organised in independent components. Second, having a large proof file is very quickly unmanageable since every single modification requires for the entire file to be reevaluated. We overcame this problem by duplicating each module by hand to take into account the two possible usages of the module inside or outside the development. This is clearly an ad-hoc and very tedious solution but it gave us the necessary modularity. The conclusion of our experiment is that modularity is clearly a must if we want to be able to handle large proof developments.

The second difficulty comes from the fact that we are using an explicit equality over polynomials. There are two main reasons why we need to define such an equality. First, our polynomials are not canonical, they may contain zero terms and we want to consider as equal those polynomials that only differ for zero terms. Second, we also want to take into account a possible equality $=_a$ over the elements of A. Using an explicit equality and not the structural one makes proofs harder mostly because we have lost the possibility to replace equals by equals. In order to regain substitutivity, we first need to prove a theorem of compatibility for each function and predicat. 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. Coq provides no real support to handle explicit equality. This is a problem not only for provers based on type theory and we do not know of any theorem prover that gives a proper solution to this problem. Still we believe that a real benefit could be gained in providing such a capability to provers.

The last difficulty concerns the automation of the proof. If we look at Figure 1, the average of 15 lines per definition or theorem shows that proofs are often reasonably short, so this seems to indicate that a lot of automation has been used. This is not the case. 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, we believe that a specific class of goals we have encountered could largely benefit from automation. In the proof development, we first introduce `terms`. After that, we define the predicate `opoly` that checks if a list of terms is ordered. We then construct the type of ordered polynomials as being `{p:term list | (opoly p)}`, i.e. the lists of terms such that the lists are ordered. In Coq, the members of a type `{p:A | (P p)}` are encoded as pairs whose first components are elements of `A` and second components are proofs that the predicate `P` holds for the first element of the pair. In a proof that manipulates ordered polynomials, it is often the case that we get several subgoals which require to prove that a polynomial is ordered. Proving such subgoals is trivial most of the time but having to repeatedly prove them becomes quickly annoying. There is no simple way to automate that task, mostly because these goals are treated as any other subgoals. It would be nice if the prover could treat them differently from the 'more substantial' goals. This is the approach developed for example by the prover PVS. Checking if an element belongs to a refined type is considered as extension of typechecking. So special heuristic are used to solve these constraints automatically.

4.3 Extracting the algorithm

Once the development is finished, not only we have the proof of correctness of the formalization 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 examples below use an instantiation of the algorithm with polynomials of dimension 6 over $\mathbb{Q}$ with the usual lexicographic order (`a` $>$ `b` $>$ `c` $>$ `d` $>$ `e` $>$ `f`). Instantiating the implementation gives us 4 functions:

1. `gen: int -> poly` creates the generators, i.e. `(gen 0) = a`, ..., `(gen 5) = f`, `(gen 6) = 1`;
2. `scal: int -> poly -> poly` multiplies the polynomial by an integer;
3. `plus: poly -> poly -> poly` adds two polynomials;
4. `mult: poly -> poly -> poly` multitplies two polynomials;
5. `buchf: poly list -> poly list` computes the Gröbner basis of a list of polynomials.

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 terminate 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 p1 =  gen 6;;
val p1 : poly = 1
```

We then construct the three n-cyclic polynomials for n=3:

```
# let r0 = (plus a (plus b c));;
val r0 : poly = a +b +c
# let r1 = (plus (mult a b) (plus (mult b c) (mult c a)));;
val r1 : poly = ab +ac +bc
# let r2 = (plus (mult a (mult b c)) (scal (-1) p1));;
val r2 : poly = abc -1
```

We can now compute the Gröbner basis:

```
# buchf [r2;r1;r0];;
- : poly list = [abc -1 ;
                 ab +ac +bc ;
                 a +b +c ;
                 -b^2c -bc^2 -1 ;
                 b^2 +bc +c^2 ;
                 c^3 -1 ]
```

Now for n=4, we have 4 polynomials:

```
# let d = gen 3;;
val d : poly = d
# 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:

```
# buchf [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 ]
```

The answers of the system to the two previous computations of Gröbner basis were immediate. This is not the case anymore for n=5. The computation had to be aborted after one hour of computation and a process of more than 100Mb! This is not too surprising: the version of the algorithm is clearly too naive to perform well on large examples.

5 Conclusion and future work

While working on this development we had clearly the feeling to be at the frontier of 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. We have described some of the problems we have encountered. The mechanism for defining functions is not yet powerful enough to allow the user to enter arbitrary functions easily. Also, the module system has to be further improved to handle parametric structures properly. Still we believe 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. Nevertheless, our claim is still that there is much to be gained in developing computer algebra algorithms in a system where it is possible to reason about them. The task of proving can always be simplified by stating a suitable set of axioms.

The work we have done on the Buchberger's algorithm is far from being finished. We first need to clean up the development in order to propose it as a reusable library of polynomials in Coq. Second, 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 a really 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. For this reason, it is far from being obvious whether the implementation error could be spotted by simple testing.

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 [6], 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 also 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, it is important to complement this initial attempt with experiments on other standard algorithms. It is clear that tackling different algorithms will give rise to other interesting problems.

Finally the correctness is not the only property we would like to mechanically derive from the algorithm. Time and space complexities are quantities we would like to be able to reason about. There have been some proposals for doing this inside a theorem prover (see for example [4]) but, as far as we know, there has been no concrete attempt to show that the proposed solutions are practical.

References

[1] Dave Bayer and Mike Stillman. Macaulay: A system for computation in algebraic geometry and commutative algebra. Available at `http://www.math.columbia.edu/-~bayer/macaulay.html`, 1994.

[2] Robert S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, 1997.

[3] Bruce W. Char, Keith O. Geddes, and Gaston H. Gonnet. *First leaves : a tutorial introduction to Maple V*. Springer-Verlag, 1992.

[4] Robert L. Constable. Expressing computational complexity in constructive type theory. volume 960 of *LNAI*. Springer-Verlag, July 1994.

[5] Robert L. Constable, Stuart F. Allen, Walter R. Cleaveland H.M. Bromley, 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*. Prentice Hall, 1986.

[6] 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.

[7] 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.

[8] Jean Charles Faugère. Résolution des systèmes d'équations algébriques. Phd thesis, Université Paris 6, 1994.

[9] Keith O. Geddes, Stephen R. Czapor, and George Labahn. *Algorithms for computer algebra.* Kluwer, 1992.

[10] Michael Gordon and Thomas Melham. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[11] John R. Harrison. Theorem proving with the real numbers. Technical Report 408, University of Cambridge Computer Laboratory, 1996.

[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] Richard D. Jenks and Robert S. Sutor. *AXIOM: the scientific computation system.* Springer-Verlag, 1992.

[15] Xavier Leroy. Objective Caml. Available at `http://pauillac.inria.fr/ocaml/`, 1997.

[16] Rob P. Nederpelt, J. Herman Geuvers, and Roel C. De Vrijer, editors. *Selected papers on Automath.* North-Holland, 1994.

[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 *Worshop on Types and Proofs for Programs.* Available at `ftp.cs.chalmers.se` in the directory `/pub/cs-reports/Bastad92/proc.ps.Z`, 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] Natarajan Shankar. *Metamathematics, Machines, and Goedel's Proof.* Cambridge University Press, 1994.

[23] Stephen Wolfram. *Mathematica : a system for doing mathematics by computer.* Addison-Wesley, 1988.

[24] Larry Wos. *The Automation of Reasoning: An Experimenter's Notebook with Otter Tutorial.* Academic Press, 1996.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399