Towards a Certified and Efficient Computing of Gröbner Bases*

J. Santiago Jorge, Víctor M. Gulías, José L. Freire, and Juan J. Sánchez

MADS - LFCIA, Dept. de Computación, Universidade da Coruña,
Campus de Elviña, s/n., 15071 A, Coruña, Spain
{sjorge, gulias, freire, juanjo}@dc.fi.udc.es

Abstract. In this paper, we present an example of the implementation and verification of a functional program. We expose an experience in developing an application in the area of symbolic computation: the computing of Gröbner basis of a set of multivariate polynomials. Our aim is the formal certification of several aspects of the program written in the functional language Caml. In addition, efficient computing of the algorithm is another issue to take into account.

1 Introduction

Certifying the correctness of a program is a difficult and expensive labor and, at the same time, it is one of the most important activities for a software engineer. Debugging and testing techniques can detect errors, but they cannot guarantee the correctness of the software. Formal methods complement those techniques assuring that some relevant property holds in the program.

This work aims to contribute to the construction of a methodology to produce certified software, i.e., we intend to find the way to strengthen two different notions: First, there is a need for formal verification of the correctness of algorithms which goes together with their construction; and second, programs are mathematical objects which can be handled using logico-mathematical tools. We present the way in which one can formally verify several aspects of the application implemented following the functional paradigm. These techniques will be exemplified by means of the verification of a program which calculates the *Gröbner basis* of a set of multivariate polynomials [1]. Here, both the certification of the program and its efficiency will be taken into consideration. We study the development of algorithms formally proved for an efficient computing. The steps are: formalization of a multivariate polynomial ring; construction of a well-founded polynomial ordering; and finally, definition of the reduction relation and Buchberger's algorithm.

Functional programming [2,3,4] has often been suggested as a suitable tool for writing programs which can be analyzed formally, and whose correctness can be assured [5]. This assertion is due to *referential transparency*, a powerful mathematical property of the functional paradigm that assures that equational

* Supported by MCyT TIC2002-02859 and Xunta de Galicia PGIDIT03PXIC10502PN.

R. Moreno Díaz et al. (Eds.): EUROCAST 2005, LNCS 3643, pp. 111–120, 2005.

reasoning makes sense. The mathematical way of proving theorems can be successfully applied to computer science. As programming language, Objective Caml [6,7] was chosen due to its efficiency and its wide coverage both in the research and academic environments.

Properties are proved by means of the formalization of theorems in an *abstract* model of *actual* code in the Coq [8,9] proof assistant and also, in a manual but exhaustive style directly applied to the final code. Programs are treated as mathematical objects and each step of those proofs is justified by means of mathematical reasoning. Significant progress has been made (see [10,11,12]) in the automated verification of the proof of Gröbner bases algorithm by proof checkers. Theorem provers assist us in proving the correctness of a program. They not only help us in the development of the proofs but also guarantee the correctness of such proofs; thus they prevent bugs that could be introduced in a hand-made certification. The logical framework Coq is an implementation of the Calculus of Inductive Constructions [13].

The paper is organized as follows. In section 2, a reusable multivariate polynomial library is certified, and a well-founded polynomial ordering is also verified. Section 3 states the correctness of the reduction relation on polynomials. Section 4 reasons about Buchberger's algorithm, and presents some results. Finally, we conclude.

2 Multivariate Polynomials Using Dependent Types

A reusable polynomial library is going to be formally verified. It will make further work in verification of polynomial algorithms less difficult. A multivariate polynomial ring over a coefficient field is our target. We not only want to prove the fundamental properties of polynomial rings, but we also search for an adequate implementation allowing efficient computing. Canonical representations of polynomials are used, and it can be decided if two polynomials are equal by studying if their representations are equal.

Although there are previous works on the formalization of multivariate polynomials by proof checkers (see [14,15]), they neither work directly with canonical representations of polynomials, nor do they use dependent types. The certification of this library has been carried out both reasoning directly about the actual functional program and also proving laws in the abstract model in Coq. Below, the main laws stated with the help of the proof assistant are presented.

A *term* in the variables $X = \{x_1, x_2, \ldots, x_n\}$ is represented by a list of the exponents of each variable.

```
type term = int list
```

In Coq, they are described as lists of fixed length with *dependent types*. The use of dependent types improves the accuracy and clarity of the specification.

```
Inductive Dlist [A: Set]: nat->Set :=
    Dnil: (Dlist A (0))
```

```
  | Dcons: (n:nat)A->(Dlist A n)->(Dlist A (S n)).
Definition term: nat->Set:= [n: nat] (Dlist nat n).
```

Dependent types provide accuracy in the specifications, which is an additional reliability. But, sometimes this precision can impose a heavy manipulation of expressions.

Two terms are multiplied by adding the respective exponents of each variable.

```
(*val mult_term : term->term->term*)
let mult_term xs ys = map2 (+) xs ys
```

This CAML code is very similiar to to the abstract model built in COQ:

```
Definition mult_term:(n:nat)(term n)->(term n)->(term n):=
 [n:nat;t,s:(term n)](map2 nat nat nat plus n t s).
```

Terms form a commutative monoid under multiplication.

```
Lemma mult_term_sym: (n: nat) (t,s: (term n))
 (mult_term n t s)=(mult_term n s t).

Lemma mult_term_assoc_l: (n:nat) (t,s,r: (term n))
 (mult_term n t (mult_term n s r))
   = (mult_term n (mult_term n t s) r).

Lemma mult_term_1_t: (n: nat; t,s: (term n))
 (null_term n t) -> (mult_term n t s)=s.
```

Other results on terms are also proved.

```
Lemma div_term_mult_term: (n: nat) (t, s: (term n))
 (div_term n (mult_term n t s) s)=t.
```

We state the lexicographical order on terms, and we show that it is an admissible order.

$$[a_1, \ldots, a_n] > [b_1, \ldots, b_n] \Leftrightarrow \exists i \text{ con } a_j = b_j \text{ for } 1 \leq j < i \text{ and } a_i > b_i$$

1. There exists a first element, $t >_{lex} 1$, $\forall t \in T_X$, $1 \neq t$

   ```
   Theorem ltlex_term_admissibility_1: (n: nat; e, t: (term n))
    (null_term n e) -> ~ (null_term n t) -> (ltlex_term n e t).
   ```

2. The ordering respects multiplication, $t >_{lex} s \Rightarrow t \cdot r >_{lex} s \cdot r$, $\forall t, s, r \in T_X$

   ```
   Theorem ltlex_term_admissibility_2: (n: nat; t, s: (term n))
    (ltlex_term n t s) -> (r:(term n))
      (ltlex_term n (mult_term n t r) (mult_term n s r)).
   ```

Monomials are represented as coefficient-term pairs. In the CAML program, absolute precision numbers (`num` library) are used as coefficients. In the model built in COQ, on the other hand, the coefficients are axiomatized. Monomials form a commutative monoid under multiplication.

```
Lemma mult_mon_sym: (m1, m2: mon)
 (eq_mon (mult_mon m1 m2) (mult_mon m2 m1)).

Lemma mult_mon_assoc_l: (m1, m2, m3: mon)
 (eq_mon (mult_mon m1 (mult_mon m2 m3))
         (mult_mon (mult_mon m1 m2) m3)).

Lemma mult_mon_1_m: (e, m: mon)
 (mon1 e) -> (eq_mon (mult_mon e m) m).
```

Polynomials are represented as lists of monomials. The representation is canonical: terms are strictly ordered by a decreasing term order, and the list contains no null monomial. Hence, two polynomials are equals if their representations are syntactically equal. As we axiomatize the coefficients in the model built in Coq, an explicit equality has to be used.

```
Inductive eq_pol  : pol->pol->Prop :=
   eq_pol_1 : (eq_pol (nil mon) (nil mon))
 | eq_pol_2 : (m1,m2:mon; p1,p2:pol)
                (eq_mon m1 m2) -> (eq_pol p1 p2) ->
                  (eq_pol (cons m1 p1) (cons m2 p2)).
```

Sometimes two different versions of a polynomial function (for instance, addition) are implemented: one efficient, the other simple. We prove they are equivalent.

Functions over polynomials always act on canonical objects to yield canonical results. Thus, on the one hand more efficient programs are obtained from an algorithmic point of view. However, on the other hand, polynomial functions become more complex because there are lots of alternatives, and consequently proofs get more complex and tedious.

```
Lemma add_pol_canonical: (p1, p2: pol)
 (canonical p1)->(canonical p2)->(canonical (add_pol p1 p2)).

Lemma mult_pol_canonical: (p1, p2: pol)
 (canonical p1)->(canonical p2)->(canonical (mult_pol p1 p2)).
```

Polynomials with addition and negation form an Abelian group.

```
Lemma add_pol_p_0: (p: pol)
 (canonical p) -> (eq_pol (add_pol p pol0) p).

Lemma add_pol_sym: (p1, p2: pol)
 (canonical p1) -> (canonical p2) ->
   (eq_pol (add_pol p1 p2) (add_pol p2 p1)).

Lemma add_pol_assoc_l: (p1, p2, p3: pol)
 (canonical p1) -> (canonical p2) -> (canonical p3) ->
   (eq_pol (add_pol p1 (add_pol p2 p3))
           (add_pol (add_pol p1 p2) p3)).
```

```
Lemma add_pol_minus_pol: (p: pol)
 (canonical p) -> (eq_pol (add_pol p (minus_pol p)) pol0).
```

With the multiplication they form a ring.

```
Lemma mult_pol_1_p: (e, p: pol)
 (canonical p) -> (pol1 e) -> (eq_pol (mult_pol e p) p).

Lemma mult_pol_sym: (p1, p2: pol)
 (canonical p1) -> (canonical p2) ->
   (eq_pol (mult_pol p1 p2) (mult_pol p2 p1)).

Lemma mult_pol_assoc_l: (p1, p2, p3: pol)
 (canonical p1) -> (canonical p2) -> (canonical p3) ->
   (eq_pol (mult_pol p1 (mult_pol p2 p3))
           (mult_pol (mult_pol p1 p2) p3)).
```

And multiplication distributes over addition.

```
Lemma mult_pol_add_mon_pol_distr: (p1, p2: pol; m: mon)
  (canonical p1) -> (canonical p2) -> (not_mon0 m) ->
      (eq_pol (mult_pol (add_mon_pol m p1) p2)
              (add_pol (mult_mon_pol m p2) (mult_pol p1 p2))).
```

2.1 Well-Founded Polynomial Ordering

The well-foundedness of the lexicographical order on terms has been verified both on the CAML program, and on the abstract model in COQ. Reasoning over the CAML code, the well-foundedness of the total degree order was also proved.

We extend the term order on monomials. With polynomials in canonical form, the monomial ordering is extended to polynomials in a straightforward

Theories	*Lines*	*Defs.*	*Laws*	*Prop.*	*Size*
Dlist	101	9	5	7.21	15K
Term	331	6	18	13.79	155K
LtlexTerm	191	1	8	21.22	146K
Coef	155	6	32	4.08	11K
Mon	171	14	18	5.34	42K
Pol	175	7	15	7.95	73K
AddMonPol	862	1	10	78.36	348K
AddPol	311	2	18	15.55	35K
MultMonPol	260	1	14	17.33	61K
MultPol	338	1	16	19.88	32K
total	2895	48	154	14.33	918K

(a) Polynomial Theories

Theories	*Lines*	*Defs.*	*Laws*	*Prop.*	*Size*
WfLtlexTerm	49	2	2	12.25	18K
WfLtlexMon	23	1	3	5.75	3K
Desc	110	0	6	18.33	18K
WfLtlexPol	98	6	7	7.54	33K
total	280	9	18	10.37	72K

(b) Well-founded Theories

Fig. 1. Quantitative Information on the Development in Coq

way. In this proof, we use a lexicografic exponentiation theory from Paulson [16] that requires monomials to be strictly ordered in a decreasing order. So, the lexicographic relation induced on *polynomials* is well-founded. In this work, we started with the lexicographic order on terms, but the development is generic. Any well-founded term ordering can be used.

Figures 1(a) and 1(b) show quantitative information on the Coq theories. The columns correspond to the number of lines of code in each theory, the number of definitions (including tactics) and the number of laws, the proportion between the number of lines and the quantity of laws and definitions (which can be used as a measure of the complexity of the theory), and the size of each compiled Coq theory, respectively.

3 Polynomial Reduction

The reduction relation on polynomials involves subtracting an appropriate multiple of one polynomial from another. Below it can be seen the Caml code.

$$red(p, q) = p - \frac{hcoef(p) \cdot hterm(p)}{hcoef(q) \cdot hterm(q)} \cdot q$$

```
(*val nred : (term->term->bool) -> pol -> pol -> pol*)
let nred gt_term f g = match (f, g) with
  ((c, t)::_, (b, s)::_) ->
    sub_pol gt_term f (mult_pol gt_term [c//b, div_term t s] g)
```

In the following example, the act of reducing p by r implies subtracting a multiple of r from p so that the head term of p is canceled: $p = 2x^2yz^3 - 7xy^{10} + z$, $r = 5xyz - 3$, the polynomial r reduces p to $p' = p - (\frac{2}{5}xz^2)r = -7xy^{10} + \frac{6}{5}xz^2 + z$.

Reduction of polynomials is not a total function because term division is not a total function. A polynomial p is reducible by q if the heading term of q divides the heading term of p. The verification of the reduction relation covers two facts:

1. $is\_reducible(p, q) \Rightarrow hterm(red(p, q)) <_{T_X} hterm(p)$
2. $is\_reducible(p, q) \Rightarrow \exists r$ such that $p = red(p, q) + r \cdot q$

Next subsection contains the certification of the two conditions above, carried out with manual proofs that treat the Caml program as a mathematical object.

3.1 Laws About Reduction

Theorem 1. *For every nonzero polynomials* `p = [(c1,t1);...;(cm,tm)]` *and* `q = [(b1,s1);...;(b1,s1)]`*, both in canonical form with respect to a term order* `gt_term`*, and such that* `is_reducible p q`*, it holds:*

```
gt_term (ht p) (ht (red gt_term p q))
```

Proof. By equational reasoning, using two previous results, and with:

```
redp = nred gt_term          divt = div_term
multp = mult_pol gt_term     multt = mult_term
subp = sub_pol gt_term
```

```
  gt_term (ht p) (ht (redp p q))
=    { 1) by definition of nred (left to right) }
  gt_term (ht p) (ht (subp p (multp [(c1/b1, divt t1 s1)] q)))
=    { 2) by definition of mult_pol (left to right) }
  gt_term (ht p) (ht (subp p (((c1/b1)*b1, multt (divt t1 s1) s1)::...)))
=    { 3) by arithmetic on num and the law (t/s)*s=t on terms }
  gt_term (ht p) (ht (subp p  ((c1,t1)::...)))
=    { 4) ht f = ht g ⇒ gt_term (ht f (ht (subp f g)) }
  true                                                              □
```

Theorem 2. *For every nonzero polynomials in canonical form with respect to a term order* `gt_term`, `p = ((c1,t1)::f')` *and* `q = ((b1,s1)::g')`, *such that* `is_reducible p q`, *it holds:*

```
p = add_pol gt_term (red gt_term p q) (mult_pol r q)
```

where: `r = [(c1/b1, div_term t1 s1)]`

Proof. By equational reasoning, using two previous results and with:

```
redp = nred gt_term          multp = mult_pol gt_term
addp = add_pol gt_term       subp = sub_pol gt_term
minusp = minus_pol           divt = div_term
```

```
  addp (redp ((c1,t1)::f') ((b1,s1)::g'))
       (multp [(c1/b1,divt t1 s1)] ((b1,s1)::g'))
=    { 1) by definition of nred (left to right) }
  addp (subp ((c1,t1)::f') (multp [(c1/b1,divt t1 s1)] ((b1,s1)::g')))
       (multp [(c1/b1,divt t1 s1)] ((b1,s1)::g'))
=    { 2) by definition of sub_pol (left to right) }
  addp (addp ((c1,t1)::f')
            (minusp (multp [(c1/b1,divt t1 s1)] ((b1,s1)::g'))))
       (multp [(c1/b1,divt t1 s1)] ((b1,s1)::g'))
=    { 3) by associativity of add_pol }
  addp ((c1,t1)::f')
       (addp (minusp (multp [(c1/b1,divt t1 s1)] ((b1,s1)::g'))))
          (multp [(c1/b1,divt t1 s1)] ((b1,s1)::g'))
=    { 4) by the law: eq_pol (add_pol p (minus_pol p)) pol0 }
  addp ((c1,t1)::f') []
=    { 5) by definition of add_pol (left to right) }
  ((c1,t1)::f')                                                     □
```

3.2 Extension of the Reduction Relation

A polynomial p is reducible modulo $Q = \{q_1, q_2, \ldots, q_m\}$, if there exists q_i such that p is reducible by q_i. We define recursively the closure of reduction.

$$full\_red(p, Q) = \begin{cases} p & \text{if } \neg(is\_reducible(p, Q)) \\ full\_red(red(p, Q), Q) & \text{otherwise} \end{cases}$$

```
(*val full_red : (term->term->bool) -> pol -> pol list -> pol*)
let rec full_red gt_term f gs = match sred gt_term f gs with
    None -> f
  | Some [] -> []
  | Some h -> full_red gt_term h gs
```

There is no infinite sequence of reductions because $<_{T_X}$ is well-founded and $hterm(red(p, Q)) <_{T_X} hterm(p)$. In addition, the result is not reducible modulo Q.

4 Buchberger's Algorithm

Buchberger's algorithm [1] is a generalization of Gaussian elimination. Given a set of polynomials, it produces another set of polynomials with the same roots and additional properties which ease the computation of those roots. The new set, called the Gröbner basis, is analogous to a triangular set of linear equations, which can be solved by substitution. The two basic operations in computing a Gröbner basis are: to eliminate one of the terms of two polynomials obtaining a new polynomial (S-polynomial), and to simplify a polynomial by subtracting multiples of other polynomials.

We implement the Buchberger's algorithm in CAML. In each recursion a pair of polynomials is selected, and the reduction of the S-polynomial is added to the set only if it is nonzero. The polynomial added is then smaller than the two selected polynomials, thus the algorithm always ends.

```
(* val buch: (term->term->bool) -> pol list -> pol list *)
let buch gt fs =
  let rec buch_aux gs = function
      [] -> gs
    | (f,g)::ps -> let h = spol gt f g in
        match full_red gt h gs with
          [] -> buch_aux gs ps
        | h' -> buch_aux (gs @ [h'])
                    (ps @ (map (fun g->(g,h')) gs)) in
    buch_aux fs (allpairs fs)
```

The function `allpairs` computes all possible pairs of the elements of a list. Function `buch_aux` is the recursive implementation of the loop of the algorithm. In each recursion, it is selected a pair and, the reduction of the S-polynomial by

	Linux 2.2.20		*Windows 98*	
	gcalc	gcalcopt	gcalc	Maple V
(1)	2.99 s.	1.05 s.	4.13 s.	12.75 s.
(2)	23.32 s.	8.25 s.	27.19 s.	69.98 s.
(3)	243.43 s.	91.12 s.	288.58 s.	519.45 s.

(a) Pentium 200MMX/48M

	Linux 2.4.22		
	gcalc	gcalcopt	Gap
(1)	1.20 s.	0.23 s.	4.10 s.
(2)	9.48 s.	1.80 s.	20.98 s.
(3)	100.59 s.	19.89 s.	117.06 s.

(b) AMD Duron 800/768M

Fig. 2. A simple benchmark of the program

gs is chosen if it is nonzero. The function always terminates because the chosen polynomial is always less than the two ones we have studied.

Figure 2 presents some measurements of the program. An execution times comparaison between our program and Maple is shown in figure 2(a). Executions have been carried out on a Pentium 200MMX/48M. Running under Linux, two different versions of our program were used: gcalc and gcalcopt, the former obtained with the bytecode compiler of Objective Caml and the latter generated with the high-performance native-code compiler [7]. Running under Windows 98 with the same hardware, we execute both Maple implementation and gcalc. See below the examples that were used employing the lexicographical order.

$$\{x^{25} - y^{25}zt,\ xz^{25} - y^{25},\ x^{25}y - z^{25}t\} \tag{1}$$
$$\{x^{50} - y^{50}zt,\ xz^{50} - y^{50},\ x^{50}y - z^{50}t\} \tag{2}$$
$$\{x^{100} - y^{100}zt,\ xz^{100} - y^{100},\ x^{100}y - z^{100}t\} \tag{3}$$

In addition, figure 2(b) presents an execution comparison between both versions of our program and GAP on a AMD Duron 800/768M running under Linux. In all cases we have repeated three times each execution and the best one was selected.

5 Conclusions

We have exposed the development of an efficient functional program for computing Gröbner bases of a set of multivariate polynomials, assuring that some relevant properties hold in the program.

We suggest to develop programs using a side-effect free language, a functional language for instance, where tools like equational reasoning make sense. Proofs of properties have been carried out both in an informal and exhaustive style (on Caml programs), and in (with the help of) the Coq proof assistant.

Elaboration of proofs is based on the syntactic structure of the program. Complex proofs are carried out with help of auxiliary laws.

The developments are kept as general as possible. Different reusable modules are implemented as, for example, an efficient multivariate polynomial library.

Program efficiency is taken into account. Sometimes two different versions of a function are implemented: one efficient, the other simple, proving their equivalence. Canonical representations that allow us to use syntactical equality

and efficient algorithms are used. Formalizing the canonical representation of polynomials is not complex, but we run into dificulties when defining operations which become more complicated causing more complex and tedious proofs.

Formal methods are not intended to provide *absolute* reliability, but to *increase* software reliability. Formal methods can be used to improve the design of systems, its efficiency, and to certify its correctness. It is often difficult to apply formal methods to a whole system. As future work, we should look for compositional techniques.

We think that the future of program verification heads for a general proposal: to obtain certified software libraries.

References

1. Buchberger, B.: An Algorithm for Finding a Basis for the Residue Class Ring of a Zero-Dimensional Polynomial Ideal. PhD thesis, Univ. of Innsbruck, Austria (1965)
2. Bird, R., Wadler, P.: Introduction to Functional Programming. Prentice Hall (1988)
3. Hudak, P.: Conception, evolution, and application of functional programming languages. ACM Computing Surveys **21** (1989)
4. Paulson, L.C.: ML for the Working Programmer. 2nd edn. Cambridge University Press (1996)
5. Jorge, J.S.: Estudio de la verificación de propiedades de programas funcionales: de las pruebas manuales al uso de asistentes de pruebas. PhD thesis, University of A Coruña, Spain (2004)
6. Weis, P., Leroy, X.: Le langage Caml. 2nd edn. Dunod (1999)
7. Leroy, X., et al.: The Objective Caml system: Documentation and User's Manual, Release 3.08. INRIA, `http://caml.inria.fr`. (2004)
8. The Coq Development Team: The Coq Proof Assistant Reference Manual, Version 7.3. INRIA, `http://coq.inria.fr`. (2002)
9. Bertot, Y., Casteran, P.: Interactive Theorem Proving and Program Development, Coq'Art: The Calculus of Inductive Constructions. Springer-Verlag (2004)
10. Théry, L.: A machine-checked implementation of Buchberger's algorithm. Journal of Automated Reasoning **26** (2001)
11. Medina-Bulo, I., Palomo-Lozano, F., Alonso-Jiménez, J.A., Ruiz-Reina, J.L.: Verified computer algebra in ACL2 (Gröbner bases computation). In: 7th International Conference on Artificial Intelligence and Symbolic Computation (AISC 2004). Volume 3249 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2004)
12. Pérez, G.: Bases de Gröbner: Desarrollo formal en Coq. PhD thesis, University of A Coruña, Spain (2005)
13. Coquand, T., Huet, G.: The calculus of constructions. Information and Computation **76** (1988)
14. Barja, J.M., Pérez, G.: Demostración en implementaciones concretas de anillos de polinomios. RSME (2000)
15. Medina-Bulo, I., Alonso-Jiménez, J.A., Palomo-Lozano, F.: Automatic verification of polynomial rings fundamental properties in ACL2. In: 2nd International Workshop on the ACL2 Theorem Prover and Its Applications. (2000)
16. Paulson, L.C.: Constructing recursion operators in intuitionistic type theory. Journal of Symbolic Computation **2** (1986)