Journal of Automated Reasoning **26:** 107–137, 2001.

A Machine-Checked Implementation of Buchberger's Algorithm ⋆

LAURENT THÉRY
INRIA, 2004 route des Lucioles, 06902 Sophia Antipolis, France. e-mail: thery@sophia.inria.fr

Abstract. We present an implementation of Buchberger's algorithm that has been proved correct within the proof assistant Coq. The implementation contains the basic algorithm plus two standard optimizations.

Key words: theorem proving, computer algebra, Gröbner bases, program verification.

1. Introduction

If we look at systems for doing mathematics on computers, there is a clear separation between computing and reasoning. Computer algebra systems are used to perform computations and implement new algorithms. Theorem-proving systems are used to formalize and reason about mathematical objects. A system that could unify these two aspects would have interesting capabilities. From the point of view of computing, this would give the possibility of stating and proving properties of algorithms. As a consequence, this would increase the confidence in the correctness of computations. From the point of view of proving, a unifying system would make it possible to tackle problems such as the formalization of mechanical devices where reasoning requires mixing proving and computing.

The aim of this paper is to show that we are not so far from having such a system. This is done by considering a nontrivial example, Buchberger's algorithm [3]. We cover all the steps from the definition of multivariate polynomials to the implementation of the algorithm and some optimizations. To do this, we use the proof assistant Coq [15]. Coq is based on type theory and has an expressive specification language that is suitable for formalizing mathematics. It also provides an extraction mechanism [20] that allows one to automatically produce from an algorithm defined in Coq an Ocaml version that can be efficiently compiled.

In the following we first give an overview of the Coq system. Then, we present our construction of polynomials and introduce the notion of Gröbner basis and its formalization in Coq. We then describe a first version of Buchberger's algorithm

⋆ A preliminary version of this work is presented in [27]. The source code of the development is available at `http://www.inria.fr/lemme/buch/`.

that we refine afterward. Finally we comment on the proof development and relate our work to others.

2. Coq

This section provides a short introduction to Coq so that the formalization presented in the next sections can be understood. For a more complete introduction, we refer the reader to [15].

Coq is a prover based on type theory. It uses the isomorphism of Curry–Howard and identifies propositions with types and proofs with terms. This means that in order to prove a proposition, we exhibit a term that has a given type. The lambda-calculus from which terms are built is the *Calculus of Inductive Constructions*. Two basic types, *Prop* and *Set*, represent respectively propositions and objects on which we can compute. As an example, let us consider the term $\lambda x\colon A.\, x$. If A is of type *Prop* (A is a proposition), this term corresponds to the proof of the proposition $A \Rightarrow A$, while if A is of type *Set*, the term represents the identity function whose type is $A \to A$.* Since Coq proposes a rich notion of type, we can go one step further in our example and construct the closed term $\lambda A\colon \mathit{Prop}.\, \lambda x\colon A.\, x$. Its type is then $\forall A\colon \mathit{Prop}.\, A \Rightarrow A$.

An important feature of the logic is that it is intuitionistic. This means that the excluded middle $\forall P\colon \mathit{Prop}.\, P \vee \neg P$ is not a valid theorem in Coq. An interesting consequence is that proving the proposition $\forall x.\exists y.P(x, y)$ in Coq is equivalent to giving an algorithm that takes an x and returns a y such that $P(x, y)$. To rephrase it in our context, proving in Coq the existence of Gröbner bases is equivalent to exhibiting an algorithm that computes them.

2.1. DEFINITIONS

New constants are introduced into Coq with the command **Definition**. For example, the previous term can be associated to the constant taut_0 by the command line

Definition $\mathit{taut}_0 : \forall A\colon \mathit{Prop}.\, A \Rightarrow A := \lambda A\colon \mathit{Prop}.\, \lambda x\colon A.\, x.$

Note that it is possible not to give the value of the constant immediately. In that case we write

Definition $\mathit{taut}_0 : \forall A\colon \mathit{Prop}.\, A \Rightarrow A.$

The system will then have to be directed to build the appropriate witness. The same definition mechanism is used with the commands **Theorem** and **Lemma** to introduce theorems and lemmas, respectively.

A development can be parameterized by using local definitions. In that case, we use the commands **Variable** and **Hypothesis**. An equivalent definition of taut_0 is

* We make the syntactic distinction between function type $\to$ and propositional implication $\Rightarrow$ only for readability. In Coq there is no difference.

Variable A: $Prop$.
Definition $taut_0 : A \Rightarrow A := \lambda x{:}\, A.\, x$.

2.2. INDUCTIVE DEFINITIONS

Coq supports inductive definitions that can be used to define new objects. For example, polymorphic lists are defined as follows:

Inductive $list[A : Set] : Set :=$
$\quad nil : (list\ A)$
$\mid \quad cons : A \to (list\ A) \to (list\ A)$.

This definition introduces a new type *list* and two new constants *nil* and *cons*. The argument between square brackets is a parameter. With this parameter taken into account, the effective types of the two constants are $\forall A : Set.\ (list\ A)$ and $\forall A : Set.\ A \to (list\ A) \to (list\ A)$. In the following, we use the Prolog notation to denote list: [] should be understood as $(nil\ A)$ and $[x, y|L]$ as $(cons\ A\ x\ (cons\ A\ y\ L))$.

For each inductive type definition, the system derives two induction principles, one for *Prop* and one for *Set*. For our type *list* we get

Definition $listInd : \forall A{:}\, Set.\, \forall P{:}\, (list\ A) \to Prop.$
$(P\ []) \Rightarrow (\forall a{:}\, A. \forall L{:}\, (list\ A).\ (P\ L) \Rightarrow (P\ [a|L])) \Rightarrow \forall L{:}\, (list\ A).\ (P\ L)$.
Definition $listRec : \forall A{:}\, Set.\, \forall P{:}\, (list\ A) \to Set.$
$(P\ []) \to (\forall a{:}\, A. \forall L{:}\, (list\ A).\ (P\ L) \to (P\ [a|L])) \to \forall L{:}\, (list\ A).\ (P\ L)$.

The first definition may be used to prove properties over lists, while the second one may be used to define functions over lists.

It is also possible to define predicates inductively. In that case only the induction principle on *Prop* is derived. For example, the predicate that expresses that an element belongs to a list can be written as

Inductive $in[A : Set] : A \to (list\ A) \to Prop :=$
$\quad inHead : \forall a{:}\, A. \forall L{:}\, (list\ A).\ (in\ A\ a\ [a|L])$
$\mid \quad inTail : \forall a, b{:}\, A. \forall L{:}\, (list\ A).\ (in\ A\ a\ L) \Rightarrow (in\ A\ a\ [b|L])$.

We use this definition mechanism intensively even when the predicate is not inductive. This is mostly a question of personal style. The main benefit is to provide a uniform treatment for predicates.

In the following, we take some liberties with the syntax of Coq.* We hide the names of the constructors of inductive predicates. We also try to follow the usual notation for functions. Given a binary operator *op*, most of the time we use the notation $op(x, y)$ instead of $(op\ x\ y)$, or $x\ op\ y$ if the operator is infix. We also allow overloading of names. With these conventions, the previous definition becomes

* Note that all our syntactic conventions can be obtained with the pretty-printing facilities of the interface CtCoq [2] that we have been using for our development.

Inductive $in[A : Set]: A \to (list\, A) \to Prop :=$
$\quad \forall a\colon A.\, \forall L\colon (list\, A).\, a\ in\ [a|L]$
$\mid \quad \forall a, b\colon A.\, \forall L\colon (list\, A).\, a\ in\ L \Rightarrow a\ in\ [b|L].$

2.3. RECURSIVE FUNCTIONS

In Coq the definition of functions is similar to what is proposed in functional languages à la ML. Nevertheless, Coq requires a proof of the termination of all the functions that are defined. The calculus of inductive constructions has a fixpoint operator that requires one argument in the recursive calls to be structurally decreasing. For example, if we define the function that appends two lists, we get

Fixpoint $+[A : Set;\ L :\ (list\ A)] : (list\ A) \to (list\ A) := \lambda M\colon (list\ A).$
$\quad$ **Cases** L **of**
$\qquad\qquad [] \Longrightarrow M$
$\qquad \mid [x|L_1] \Longrightarrow [x|L_1 + M]$
$\quad$ **end**.

An arbitrary syntactic convention in Coq is that it is the last parameter of the fixpoint that decreases in recursive calls. In our example, it is the first argument L that is decreasing. This explains why the second argument M is introduced via a lambda-abstraction. An important property of functions defined by fixpoint is that they can be evaluated inside Coq by reduction.

For functions whose termination cannot be derived by a structural argument, we use the property of well-foundedness that is defined in Coq as

Variable $A\colon Set$.
Variable $R\colon A \to A \to Prop$.
Inductive $Acc\colon A \to Prop :=$
$\quad \forall x\colon A.\, (\forall y\colon A.\, y\ R\ x \Rightarrow Acc(y)) \Rightarrow Acc(x).$
Definition $wellFounded\colon Prop := \forall a\colon A.\, Acc(a).$

From this, a general principle of function definition is derived:

Definition $wellFoundedInduction : \forall A\colon Set.\, \forall P\colon A \to Set.$
$\quad \forall R\colon A \to A \to Set.\, wellFounded(R) \to$
$\qquad (\forall x\colon A.\, (\forall y\colon A.\, y\ R\ x \to (P\ y)) \to (P\ x)) \to$
$\qquad\quad \forall a\colon A.\, (P\ a).$

This definition states that, in order to define a function over A, we simply need to give a function that computes the value for an arbitrary x of A just by knowing the values for all the y smaller than x with respect to R. For the moment, proving properties of functions directly from their definition when using this principle is difficult in Coq. We use a standard technique to overcome this problem. Instead of reasoning on the definition, we prove properties on the graph of the function. For each function f, we define the predicate P_f such that $P_f(x, y)$ if and only if

$y = f(x)$. This predicate can be seen as the Prolog version of the function. We prove properties on the predicate P_f and then lift them to the function f.

To simplify the presentation, we make no distinction between the two ways of defining functions in the next sections. We always introduce functions with the command **Fixpoint**. When needed, we provide the well-founded relation that ensures termination explicitly.

2.4. SPECIFIC CONSTRUCTIONS

In the following we make use of three predefined notions: equality, sum, and subset. The equality of Coq is the one of Leibniz. Two terms are equal if they are structurally equal. Equality is defined in Coq as the smallest relation that is reflexive:

Inductive $== [A : Set]: A \to A \to Prop :=$
$\quad eqRefl : \forall a{:}\, A.\, a == a$

The sum is defined in Coq as

Inductive $\{\,\} + \{\,\}\ [A, B : Prop]: Set :=$
$\quad left : A \to \{A\} + \{B\}$
$\mid \quad right : B \to \{A\} + \{B\}.$

It is the exact counterpart in *Set* of the definition of disjunction for propositions. $\{A\} + \{B\}$ can be read as *A or B*, knowing that an object of type $\{A\} + \{B\}$ is either a proof of A (*left*) or a proof of B (*right*). We mainly use the sum to define test functions on predicates. For example, the equality test for natural numbers is defined as

Definition $NatDec : \forall a, b{:}\, Nat.\, \{a == b\} + \{\neg(a == b)\}.$

To introduce more than two elements in a test, we also need another sum type:

Inductive $+\{\,\}\ [A : Set;\ B : Prop]: Set :=$
$\quad inleft : A \to A + \{B\}$
$\mid \quad inright : B \to A + \{B\}.$

Now we can define the comparison function for natural numbers as

Definition $LtDec : \forall a, b{:}\, Nat.\, \{a < b\} + \{b < a\} + \{a == b\}.$

As sums are of type *Set*, we can use them in function definitions. For example, the function that computes the maximum of two natural numbers is defined as

Definition $max : Nat \to Nat \to Nat := \lambda a, b{:}\, Nat.$
$\quad$ **Cases** $LtDec(a, b)$ **of**
$\quad\quad inleft(left) \Longrightarrow b$
$\quad \mid inleft(right) \Longrightarrow a$
$\quad \mid \quad inright \Longrightarrow b$
$\quad$ **end**.

Note that in the pattern matching we can omit proof terms.

To make the definition of case expressions more readable, we use the following notation:

Definition $max: Nat \to Nat \to Nat := \lambda a, b: Nat.$
  Cases $LtDec(a, b)$ **of**
    $\{a < b\} \Longrightarrow b$
  | $\{b < a\} \Longrightarrow a$
  | $\{a == b\} \Longrightarrow b$
  end.

Finally we represent subsets using the following type:

Inductive { : | } $[A : Set;\ P : A \to Prop]: Set :=$
  $exist : \forall x: A.\ (P\ x) \to \{x : A \mid (P\ x)\}$

Given a unary predicate P on A, we often need to consider the subset of A that is composed of those elements of A that verify P. Using the previous definition, we represent an element of the subset by a pair containing the element plus a proof that it verifies P. For example, we define the subset of the nonempty polymorphic lists as

Definition $nZlist: Set \to Set := \lambda A: Set.\ \{L : (list\ A) \mid \neg(L == [])\}$.

3. Polynomials and Their Operations

Before constructing our polynomials, we first need to introduce some terminology.* A polynomial is composed of a set of *terms*. Each term is composed of a *coefficient* and a *monomial*. For example, the polynomial over $\mathbb{Q}$,

$$3x^2y + 5y + 3,$$

is composed of three terms, where the coefficient of the first term is 3 and its monomial is x^2y.

3.1. POLYNOMIALS AS ORDERED LISTS

The first decision we have taken in our construction of polynomials has been to abstract terms. It is a simple shortcut to get a construction that is generic not only with respect to the field of coefficients but also with respect to the monomial representation. Here are the main definitions of our terms:

* We depart from [11] and use Bourbaki's terminology.

Variable $term: Set$.
Variable $1: term$.
Variable $zeroP: term \to Prop$.
Variable $=, =_m, <_m : term \to term \to Prop$.
Variable $-: term \to term$.
Variable $+, -, *, \hat{}: term \to term \to term$.
Variable $/: \forall a, b: term.\, \neg zeroP(b) \to term$.

These lines deserve some comments. We have an explicit unit term, but no null term. This is replaced by the predicate *zeroP*. We also have the predicate $=$ to represent equality between terms. We have chosen to have these two predicates to mimic what is done in computer algebra systems, where the zero test and the equality can be implemented by two different algorithms. This is reflected by the two test functions:

Variable $zeroPDec: \forall a: term.\, \{zeroP(a)\} + \{\neg zeroP(a)\}$.
Variable $=Dec: \forall a, b: term.\, \{a = b\} + \{\neg(a = b)\}$.

The predicate $=_m$ represents the equality of the monomial parts of two terms and $<_m$ an arbitrary monomial ordering. The comparison function is given by

Variable $<_m Dec: \forall a, b: term.\, \{a <_m b\} + \{b <_m a\} + \{a =_m b\}$.

We define addition and subtraction of terms as total functions while these functions make sense only if they are applied to terms with the same monomial. Each hypothesis on these operations needs then to be guarded with the adequate condition on monomials. For example, the commutativity of addition is stated as

Hypothesis $+Com: \forall x, y: term.\, x =_m y \Rightarrow x + y = y + x$.

A similar problem occurs for the division but this time we also have to avoid dividing by a null coefficient. This is why the division takes not only its two arguments but also a proof that the second argument is not zero. Since the division carries a proof, we also need to have the hypothesis stating that the values of the function are independent of the proof argument:

Hypothesis $/Irr: \forall x, y: term.\, \forall Z_1, Z_2: \neg zeroP(y).\, x/_{Z_1} y == x/_{Z_2} y$.

In the following we hide the proof argument of the division. Finally, $x\hat{}y$ represents a least common multiplier of the terms x and y. The results of all these definitions and properties is a relatively heavy axiomatization consisting of 75 axioms. In practice, proving that a given term structure verifies these axioms simply consists in lifting properties of coefficients and monomials to terms.

Our polynomials are represented by ordered lists of terms containing no null terms and with the head term being the biggest. We first define the property of being ordered with the predicate $\mathcal{O}$:*

* In fact we are using an existing library of Coq where an equivalent predicate is defined that, for technical reasons, checks the ordering starting from the tail of a list.

Inductive $\mathcal{O}$: (*list term*) $\rightarrow$ *Prop* :=
  $\mathcal{O}([])$
| $\forall x$: *term*. $\mathcal{O}([x])$
| $\forall x, y$: *term*. $\forall L$: (*list term*). $y <_m x \Rightarrow \mathcal{O}([y|L]) \Rightarrow \mathcal{O}([x, y|L])$.

and the property of not carrying null terms by the predicate *nZ*:

Inductive *nZ*: (*list term*) $\rightarrow$ *Prop* :=
  nZ([])
| $\forall x$: *term*. $\forall L$: (*list term*). $\neg$*zeroP*(x) $\Rightarrow$ *nZ*(L) $\Rightarrow$ *nZ*($[x|L]$).

The predicate $\mathcal{C}$ defines the canonicity of a list of terms by combining the two properties above:

Inductive $\mathcal{C}$: (*list term*) $\rightarrow$ *Prop* :=
  $\forall L$: (*list term*). $\mathcal{O}(L) \Rightarrow$ *nZ*(L) $\Rightarrow \mathcal{C}(L)$.

We can now define our polynomials as canonical lists of terms:

Definition *poly*: *Set* := $\{p$: (*list term*) $| \mathcal{C}(p)\}$.

In the following we make use of the constructor and the projections of this new type (we do not give the definitions explicitly):

Definition *mkP* : $\forall p$: (*list term*). $\mathcal{C}(p) \rightarrow$ *poly*.
Definition *gL* : *poly* $\rightarrow$ (*list term*).
Definition *gC* : $\forall p$: *poly*. $\mathcal{C}$(*gL*(p)).

3.2. EQUALITY

We define two polynomials as being equal if their lists of terms are equal:

Inductive = : (*list term*) $\rightarrow$ (*list term*) $\rightarrow$ *Prop* :=
  [] = []
| $\forall x, y$: *term*. $\forall L, M$: (*list term*). $x = y \Rightarrow L = M \Rightarrow [x|L] = [y|M]$.

Definition = : *poly* $\rightarrow$ *poly* $\rightarrow$ *Prop* := $\lambda x, y$: *poly*. *gL*(x) = *gL*(y).

It is easy to prove that if = is a relation of equivalence (reflexive, symmetric, and transitive) over terms, so is = over polynomials.

3.3. ADDITION

The definition is by a simple case analysis on the two lists where we take care not to create null terms:

Fixpoint + [L, M: (*list term*)] : (*list term*) :=
  Cases L M **of**
    [] M $\Longrightarrow M$
  | L [] $\Longrightarrow L$

$|\,[x|L_1]\,[y|M_1] \Longrightarrow$ **Cases** $<_m Dec(x, y)$ **of**
$\{x <_m y\} \Longrightarrow [y|L + M_1]$
$|\,\{y <_m x\} \Longrightarrow [x|L_1 + M]$
$|\,\{x =_m y\} \Longrightarrow$ **let** $z = x + y$ **in**
Cases $zeroPDec(z)$ **of**
$\{zeroP(z)\} \Longrightarrow L_1 + M_1$
$|\,\{\neg zeroP(z)\} \Longrightarrow [z|L_1 + M_1]$
end
end
end.

To prove the termination of the function, we use the fact that the sum of the lengths of the two lists in the recursive calls always decreases.

From this definition, it is relatively easy to prove that addition preserves canonicity, so we can lift addition from lists to polynomials:

Theorem $+\mathcal{C}$: $\forall L, M$: $(list\ term).\ \mathcal{C}(L) \Rightarrow \mathcal{C}(M) \Rightarrow \mathcal{C}(L + M)$.

Definition $+$: $poly \rightarrow poly \rightarrow poly :=$
$\lambda p, q$: $poly.\ mkP(gL(p) + gL(q), +\mathcal{C}(gL(p), gL(q), g\mathcal{C}(p), g\mathcal{C}(q)))$.

Proving properties of addition is rather tedious as the function is defined by nested cases. For example, the proof of associativity

Theorem $+Assoc$: $\forall p, q, r$: $poly.\ (p + q) + r = p + (q + r)$.

gives rise to 25 different cases.

Moreover, we have an explicit equality between polynomials, so to get the replacement of equals by equals for polynomials, we need to prove compatibility theorems for each new function. For addition we prove that

Theorem $+Comp$: $\forall p, q, r, s$: $poly.\ p = r \Rightarrow q = s \Rightarrow p + q = r + s$.

3.4. MULTIPLICATION BY A TERM

The multiplication of a polynomial by a term is defined as follows:

Fixpoint $.[x : term;\ L : (list\ term)] : (list\ term) :=$
Cases L **of**
$[] \Longrightarrow []$
$|\,[y|L_1] \Longrightarrow [x * y \,|\, x.L_1]$
end.

To get the theorem of canonicity, we suppose that the set of coefficients is an integral domain, that is, a product is null if and only if one of its components is null. So we have

Theorem $.\mathcal{C}$: $\forall x$: $term.\ \forall L$: $(list\ term).\ \neg zeroP(x) \Rightarrow \mathcal{C}(L) \Rightarrow \mathcal{C}(x.L)$.

We have two ways of lifting this function to polynomial: either we keep the nonzero condition as argument of the function, or we do a case analysis inside the function to check whether the term is zero or not. Since our polynomials are canonical, we are mostly interested in the first way of lifting:

Definition $. : \forall x\colon term.\, \neg zeroP(x) \to poly \to poly$.

As we do already for the division, from now on we omit the proof argument in the application of the multiplication by a term.

3.5. SUBTRACTION

Defining subtraction in terms of addition and multiplication by a term would be fine from the proving point of view, but algorithmically it would mean that a subtraction would cost an addition plus a multiplication. So we properly define subtraction in a way similar to addition in Section 3.3 and prove the relation between addition and subtraction:

Theorem $-Rel + : \forall p, q\colon poly.\, p - q = p + (-1).q$.

3.6. MULTIPLICATION

For the proof of correctness of the algorithm we do not need to introduce the multiplication of two polynomials. Nevertheless, multiplication is needed in order to prove one of the optimizations. Its definition is as follows:

Fixpoint $*[L, M : (list\ term)] : (list\ term) :=$
  Cases L **of**
    $[] \Longrightarrow []$
  $|\,[x|L_1] \Longrightarrow x.M + L_1 * M$
  end.

Note that the ordering is preserved because the order is a monomial ordering so it is compatible with multiplication. Also the fact that L and M are canonical ensures that $x.M$ does not generate null terms.

3.7. LEXICOGRAPHIC ORDERING

The lexicographic ordering plays a special role in Buchberger's algorithm. As we did for equality, we first introduce it at the level of lists of terms and then lift it to polynomials:

Inductive $<$: (*list term*) $\rightarrow$ (*list term*) $\rightarrow$ *Prop* :=
$\forall x$: *term*. $\forall L$: (*list term*). [] $< [x|L]$
| $\forall x, y$: *term*. $\forall L, M$: (*list term*). $x <_m y \Rightarrow [x|L] < [y|M]$
| $\forall x, y$: *term*. $\forall L, M$: (*list term*). $x =_m y \Rightarrow L < M \Rightarrow [x|L] < [y|M]$.

Definition $<$: *poly* $\rightarrow$ *poly* $\rightarrow$ *Prop* := $\lambda x, y$: *poly*. $gL(x) < gL(y)$.

Polynomials are ordered lists, so what we have defined is a lexicographic exponential order [21]. The order on terms is a monomial ordering, so it is well-founded. It follows that the order on polynomials we have just defined is also well-founded.

The definitions of functions and predicates over polynomials follow the same pattern. We first define them at the level of lists and prove a theorem of canonicity for the functions. Then we lift the definitions to polynomials. In the following, in order to shorten the presentation, we directly provide the definitions at the level of polynomials. We use the notation 0 to denote the null polynomial and $a \stackrel{>}{+} p$ to denote the polynomials with head term a and tail p. With this convention we can state the two inductive theorems that are derived from the construction of polynomials and the order:

Theorem *StructInd*: $\forall P$: *poly* $\rightarrow$ *Prop*.
$P(0) \Rightarrow (\forall a$: *term*. $\forall p$: *poly*. $P(p) \Rightarrow P(a \stackrel{>}{+} p)) \Rightarrow \forall p$: *poly*. $P(p)$.

Theorem *<Ind*: $\forall P$: *poly* $\rightarrow$ *Prop*.
$(\forall p$: *poly*. $(\forall q$: *poly*. $q < p \Rightarrow P(q)) \Rightarrow P(p)) \Rightarrow \forall p$: *poly*. $P(p)$.

4. Gröbner Bases

In this section we introduce the notions of polynomial ideals, reduction, and Gröbner basis. We then show how the property of a basis to be a Gröbner basis relates to the confluence of the reduction. Most of the notation that we are using is taken from [11].

4.1. POLYNOMIAL IDEALS

An ideal I in a commutative ring R is a subset of R that is closed under addition by elements of I and multiplication by elements of R. Given a set of polynomials S, it is possible to generate an ideal $\langle S \rangle$ that is composed of all the polynomials $p = \sum_{i<k} t_i . p_i$, where k is an integer, $t_1, \ldots, t_{k-1}$ are terms, and $p_1, \ldots, p_{k-1}$ are elements of S. A set of polynomials S is said to be a *basis* of an ideal I if and only if $\langle S \rangle = I$.

In our formalization we translate sets of polynomials into lists of polynomials. We avoid defining the notion of ideal and generated ideal by introducing a predicate that characterizes the polynomials that are combinations of a list of polynomials:

Inductive $Cb[S : (list\ poly)]: poly \to Prop :=$
$\quad Cb_S(0)$
$|\quad \forall a: term.\, \forall p, q, r: poly.$
$\quad\quad \neg zeroP(a) \Rightarrow q\ in\ S \Rightarrow Cb_S(p) \Rightarrow r = a.q + p \Rightarrow Cb_S(r).$

Theorems that would be stated as $\forall p \in \langle S \rangle.\ P(p)$ are translated into $\forall p: poly.\, Cb_S(p) \Rightarrow P(p)$. Using this definition, we derive the following two theorems easily:

Theorem $Cb^+: \forall S: (list\ poly).\, \forall p, q: poly.$
$\quad Cb_S(p) \Rightarrow Cb_S(q) \Rightarrow Cb_S(p + q).$

Theorem $Cb^*: \forall S: (list\ poly).\, \forall p, q: poly.\, Cb_S(q) \Rightarrow Cb_S(p * q).$

4.2. REDUCTION

A polynomial p is reduced with respect to a set of polynomials S. If p reduces to q, we write $p \to_S q$. Reducing consists in deleting a term of p using a polynomial in S. As an example, we consider the polynomial $p = 3x^2y^2 + 2z^3$ and the set of polynomials $S = \{xy + 1, z + 2\}$. We can delete either the term $3x^2y^2$ in p by subtracting the first polynomial of S multiplied by $3xy$, or the term $2z^3$ by subtracting the second polynomial of S multiplied by $2z^2$. We get respectively $p \to_S 2z^3 - 3xy$ and $p \to_S 3x^2y^2 - 4z^2$.

To formalize reduction, we first define the predicate of divisibility for nonzero terms:

Inductive $divP: term \to term \to Prop :=$
$\quad \forall a, b: term.\, \neg zeroP(a) \Rightarrow \neg zeroP(b) \Rightarrow a = (a/b) * b \Rightarrow divP(a, b).$

With this definition, $divP(a, b)$ should be read as b divides a. We are ready to define a step of reduction.

Inductive $\to [S : (list\ poly)]: poly \to poly \to Prop :=$
$\quad \forall a, b: term.\, \neg zeroP(b) \Rightarrow \forall p, q, r: poly.$
$\quad\quad b \hat{+} q\ in\ S \Rightarrow divP(a, b) \Rightarrow r = p - (a/b).q \Rightarrow a \hat{+} p \to_S r$
$|\quad \forall a, b: term.\, \forall p, q: poly.\, a = b \Rightarrow p \to_S q \Rightarrow a \hat{+} p \to_S b \hat{+} q.$

Note that we make use of the equality explicitly to ensure that our definition is compatible with equality. We also define the notion of irreducibility:

Inductive $irreducible[S : (list\ poly)]: poly \to Prop :=$
$\quad \forall p: poly.\, (\forall q: poly.\, \neg(p \to_S q)) \Rightarrow irreducible_S(p).$

Finally, we define the reflexive transitive closure of the reduction and the reduction till irreducibility:

Inductive $\to^+ [S : (list\ poly)]: poly \to poly \to Prop :=$
$\quad \forall p, q: poly.\, p = q \Rightarrow p \to_S^+ q$
$|\quad \forall p, q, r: poly.\, p \to_S q \Rightarrow q \to_S^+ r \Rightarrow p \to_S^+ r.$

Inductive $\rightarrow^*$ [S : (*list poly*)]: *poly* $\rightarrow$ *poly* $\rightarrow$ *Prop* :=
$\forall p, q$: *poly*. $p \rightarrow_S^+ q \Rightarrow$ *irreducible*$_S(q) \Rightarrow p \rightarrow_S^* q$.

An important property of the reduction is that it does not change the membership:

Lemma *RedCb*:
$\forall S$: (*list poly*). $\forall p, q$: *poly*. $p \rightarrow_S q \Rightarrow (Cb_S(p) \iff Cb_S(q))$.

A second property is that the reduced polynomial is always strictly smaller than the one it comes from:

Lemma *RedLess*: $\forall S$: (*list poly*). $\forall p, q$: *poly*. $p \rightarrow_S q \Rightarrow q < p$.

These two lemmas are direct consequences of the definition of the reduction.

Having defined the reduction as a predicate makes it possible to prove properties that are independent of a particular reduction strategy. A reduction strategy *reducef* is just a function of type* (*list poly*) $\rightarrow$ *poly* $\rightarrow$ *poly* such that $\forall S$: (*list poly*). $\forall p$: *poly*. $p \rightarrow^*$ *reducef*$_S(p)$.

4.3. GRÖBNER BASES AND CONFLUENCE

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. A set of polynomials S is a *Gröbner basis* if and only if $\forall p$: *poly*. $p \in \langle S \rangle \iff p \rightarrow_S^* 0$. In other words, the only irreducible polynomial of the ideal generated by a Gröbner basis is zero. In our formalization we characterize the fact of being a Gröbner basis as follows:

Inductive *Gröbner*[S : (*list poly*)]: *Prop* :=
$(\forall p$: *poly*. $Cb_S(p) \Rightarrow p \rightarrow_S^* 0) \Rightarrow$ *Gröbner*(S).

Our definition is equivalent to the previous one, because the following lemma is derived from the theorem *RedCb*:

Lemma *Red*ImpCb*: $\forall S$: (*list poly*). $\forall p$: *poly*. $p \rightarrow_S^* 0 \Rightarrow Cb_S(p)$.

The example at the beginning of Section 4.2 shows that there is not a unique way of reducing a polynomial till irreducibility. But as all strategies terminate, to check whether a given polynomial belongs to an ideal generated by a Gröbner basis, one simply needs to reduce it to an irreducible polynomial and then test whether the result is zero.

The property of being a Gröbner basis can be related to the property of being confluent:

Inductive *Confluent*[S : (*list poly*)]: *Prop* :=
$(\forall p, q, r$: *poly*. $p \rightarrow_S^* q \Rightarrow p \rightarrow_S^* r \Rightarrow q = r) \Rightarrow$ *Confluent*(S).

* This type characterizes only terminating strategies, but the theorem *RedLess* and the fact that $<$ is well-founded ensure that all strategies terminate.

Theorem *ConfImpGröb*: $\forall S$: (*list poly*). *Confluent*$(S) \Rightarrow$ *Gröbner*(S).

To prove the previous theorem, we use two key lemmas:

Lemma *RedCompMinus*: $\forall S$: (*list poly*). $\forall p, q, r$: *poly*.
$p - q \rightarrow_S r \Rightarrow \exists p_1, q_1$: *poly*. $p \rightarrow_S^+ p_1 \wedge q \rightarrow_S^+ q_1 \wedge r = p_1 - q_1$.

Lemma *Red*$^+$*Minus0*:
$\forall S$: (*list poly*). $\forall p, q$: *poly*. $p - q \rightarrow_S^+ 0 \Rightarrow \exists r$: *poly*. $p \rightarrow_S^+ r \wedge q \rightarrow_S^+ r$.

To prove the first lemma we just look at the term in $p-q$ that has been reduced 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 proof of the theorem *ConfImpGröb* is given in Appendix A. It uses the existence of a reduction strategy till irreducibility *reducef*. In our development, this existence is proved by introducing the usual head reduction strategy.

5. Buchberger's Algorithm

In the preceding section we have shown that the property of being a Gröbner basis is related to the confluence of the reduction. Buchberger's contribution was to give an explicit algorithm for computing a Gröbner basis corresponding to an initial set of polynomials. The algorithm can be seen as a special case of Knuth–Bendix completion [17]. We first introduce the notion of spolynomials that plays the role of critical pairs and then present the algorithm and give its proof of correctness.

5.1. SPOLYNOMIALS

To define what spolynomials are, we use the function $\hat{}$ that computes a least common multiplier of two terms. Then we define the function *Spoly* as follows:

Definition *Spoly*: *poly* $\rightarrow$ *poly* $\rightarrow$ *poly* $:= \lambda p, q$: *poly*.
Cases $p \quad q$ **of**
$\quad 0 \quad q \quad \Longrightarrow 0$
$\mid \quad p \quad 0 \quad \Longrightarrow 0$
$\mid x \stackrel{>}{+} p_1 \quad y \stackrel{>}{+} q_1 \Longrightarrow$ **let** $z = x\hat{}y$ **in** $(z/x).p_1 - (z/y).q_1$
end.

To gain some intuition of what spolynomials represent, just consider two polynomials $p = t_1 \stackrel{>}{+} p_1$ and $q = t_2 \stackrel{>}{+} p_2$. The polynomial $t_1\hat{}t_2$ represents an "atomic" source of divergence, as it can be reduced by both polynomials p and q:

$$t_1\hat{}t_2 \rightarrow_{[p]} q_1 = -((t_1\hat{}t_2)/t_1).p_1$$

$$t_1\hat{}t_2 \rightarrow_{[q]} q_2 = -((t_1\hat{}t_2)/t_2).p_2$$

It is easy to check that $Spoly(p, q) = q_2 - q_1$. Now if we look at the theorem *Red*$^+$*Minus0*, we can see that the reducibility of the spolynomial to zero removes the divergence. This is formalized by the following theorem:

Inductive *SpolyP*[S : (*list poly*)]: *Prop* :=
($\forall p, q$: *poly*. p *in* $S \Rightarrow q$ *in* $S \Rightarrow Spoly(p, q) \rightarrow_S^* 0) \Rightarrow SpolyP(S)$.

Theorem *SpolyImpConf*: $\forall S$: (*list poly*). $SpolyP(S) \Rightarrow Confluent(S)$.

To prove it, we need an extra key lemma:

Lemma *RedDistMinus*: $\forall S$: (*list poly*). $\forall p, q, r$: *poly*.
$p \rightarrow_S q \Rightarrow \exists s$: *poly*. $p - r \rightarrow_S^+ s \wedge q - r \rightarrow_S^+ s$.

The witness is given by repeating if possible in $p - r$ and $q - r$ the same term cancellation as in p. The proof of the theorem *SpolyImpConf* is given in Appendix B.

The theorem *SpolyImpConf* combined with the theorem *ConfImpGröb* gives our central theorem:

Theorem *SpolyImpGröb*: $\forall S$: (*list poly*). $SpolyP(S) \Rightarrow Gröbner(S)$.

Note that we have, in fact, an equivalence, because the converse is a consequence of the two lemmas:

Lemma *InCb*: $\forall S$: (*list poly*). $\forall p$: *poly*. p *in* $S \Rightarrow Cb_S(p)$.

Lemma *SpolyCb*: $\forall S$: (*list poly*). $\forall p, q$: *poly*.
$Cb_S(p) \Rightarrow Cb_S(q) \Rightarrow Cb_S(Spoly(p, q))$.

5.2. THE NAÏVE ALGORITHM

We can now present our first version of Buchberger's algorithm. It is composed of four function definitions. The first function *SpolyL* adds all the spolynomials composed of a polynomial and the elements of a list to another list:

Fixpoint *SpolyL* [p : *poly*; S, T : (*list poly*)] : (*list poly*) :=
  Cases T **of**
        $[] \Longrightarrow S$
    $|\,[q|T_1] \Longrightarrow [Spoly(p, q)|SpolyL(p, S, T_1)]$
  end.

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

Fixpoint *SpolyProd* [S : (*list poly*)] : (*list poly*) :=
  Cases S **of**
        $[] \Longrightarrow []$
    $|\,[p|T] \Longrightarrow SpolyL(p, SpolyProd(T), T)$
  end.

The function *Buchf* does the completion. It takes two arguments. The first one is the basis to be completed and the second one the candidates to complete the basis:

Fixpoint *Buchf* $[S, T : (list\ poly)] : (list\ poly) :=$
  Cases T **of**
      $[] \Longrightarrow S$
    $|\,[p|T_1] \Longrightarrow$ **let** $z = reducef_S(p)$ **in**
      Cases $zeroPDec(z)$ **of**
        $\{zeroP(z)\} \Longrightarrow Buchf(S, T_1)$
      $|\,\{\neg zeroP(z)\} \Longrightarrow Buchf([z|S], SpolyL(z, T_1, S))$
      end
  end.

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

Definition *Buch*: $(list\ poly) \to (list\ poly) :=$
  λS: $(list\ poly).\ Buchf(S, SpolyProd(S))$.

Only the termination of the function *Buchf* is problematic. To prove that this function terminates, we first define a relation $\Re$ on lists of terms and suppose that $\Re$ is well founded:

Inductive $\Re$: $(list\ term) \to (list\ term) \to Prop :=$
  $\forall x$: $term.\ \forall S$: $(list\ term).\ \neg zeroP(x) \Rightarrow$
    $(\forall y$: $term.\ y\ in\ S \Rightarrow \neg divP(x, y)) \Rightarrow [x|S]\,\Re\,S$.
Hypothesis $\Re wf$: $wellFounded(\Re)$.

If we look at the second recursive call in *Buchf*, we see that z is irreducible by S. In particular this means that its head term is not divisible by any of the head terms of polynomials in S. So the first argument of *Buchf* cannot grow indefinitely because $\Re$ is well founded. As in the first recursion the size of the second list decreases, a lexicographic product gives us the termination of the function *Buchf*.

The hypothesis $\Re wf$ is a consequence of a weak version of Dickson's lemma. This lemma states that in every infinite sequence of monomials, there exists at least one monomial M_i that divides another monomial M_j such that $i < j$. If $\Re$ were not well founded, there would be an infinite sequence $(L_i)_{i \in N}$ such that $L_{i+1}\,\Re\,L_i$. As $L_i \subset L_{i+1}$, we could build an infinite sequence of terms that would contradict Dickson's lemma.

5.3. PROOF OF CORRECTNESS

Once the function *Buch* has been defined in Coq, all we know is that this function always terminates. Its correctness can now be expressed by two theorems. The first one ensures that the result of the function *Buch* does not change the ideal:

Inductive $Stable[S, T : (list\ poly)]: Prop :=$
$(\forall p: poly.\ Cb_S(p) \iff Cb_T(p)) \Rightarrow Stable(S, T)$.

Theorem *BuchStable*: $\forall S: (list\ poly).\ Stable(S, Buch(S))$.

One direction of the equivalence is proved using the following lemma:

Lemma *InclCb*: $\forall p: poly.\ \forall S, T: (list\ poly)$.
$(\forall q: poly.\ q\ in\ S \Rightarrow q\ in\ T) \Rightarrow Cb_S(p) \Rightarrow Cb_T(p)$.

The other direction is a consequence of the lemmas *InCb*, *SpolyCb*, *RedCb*, and the lemma:

Lemma *TransCb*:
$\forall p, q: poly.\ \forall S: (list\ poly).\ Cb_{[q|S]}(p) \Rightarrow Cb_S(q) \Rightarrow Cb_S(p)$.

The second theorem states that the result of *Buch* is a Gröbner basis:

Theorem *BuchGröbner*: $\forall S: (list\ poly).\ Gröbner(Buch(S))$.

Because of the theorem *SpolyImpGröb*, we have to prove

Theorem *BuchSpoly*: $\forall S: (list\ poly).\ SpolyP(Buch(S))$.

This is proved by induction on the execution paths of *Buch*, using the monotonicity of the reduction to zero:

Lemma *InclRed**: $\forall p: poly.\ \forall S, T: (list\ poly)$.
$(\forall q: poly.\ q\ in\ S \Rightarrow q\ in\ T) \Rightarrow p \to_S^* 0 \Rightarrow p \to_T^* 0$.

Note also that the function *SpolyProd* does not generate all possible spolynomials but only a reduced set. The two lemmas

Lemma *SpolyId*: $\forall p: poly.\ Spoly(p, p) = 0$.
Lemma *SpolySym*: $\forall p, q: poly.\ Spoly(p, q) = (-1).Spoly(q, p)$.

ensure that the reduction to zero of the reduced set implies the reduction of the complete set.

6. Refining the Algorithm

Once the naïve algorithm has been proved to be correct, we can develop more elaborated versions of the algorithm. We first introduce two standard criteria that justify the optimizations we are going to apply. We then present the new algorithm to compute reduced Gröbner bases. Finally we show how the extracted program can be applied on examples.

6.1. FIRST CRITERION

Two terms are *foreign* if their product is a least common multiplier. By lifting to polynomials, two nonzero polynomials are foreign if their head terms are foreign. We represent this property by the following predicate:

Inductive *foreign*: $poly \to poly \to Prop :=$
$\forall p$: *poly*. *foreign*$(p, 0)$
| $\forall q$: *poly*. *foreign*$(0, q)$
| $\forall x, y$: *term*. $\forall p, q$: *poly*. $x\hat{}y =_m x * y \Rightarrow$ *foreign*$(x \stackrel{>}{+} p, y \stackrel{>}{+} q)$.

An important property is that the spolynomial of foreign polynomials reduces to zero:

Theorem *ForeignRed**: $\forall p, q$: *poly*.
foreign$(p, q) \Rightarrow$ *Spoly*$(p, q) \to^*_{[p,q]} 0$.

In the optimized algorithm we will avoid checking the reducibility to zero of foreign polynomials using the test function:

Definition *foreignDec* : $\forall p, q$: *poly*. $\{$*foreign*$(p, q)\} + \{\neg$*foreign*$(p, q)\}$.

The proof of the theorem *ForeignRed** is rather technical. We use three key lemmas:

Lemma *MultRed**: $\forall p, q$: *poly*. $p * q \to^*_{[q]} 0$.

Lemma *ForeignTerm*: $\forall a, b, c, d$: *term*.
foreign$(a, b) \Rightarrow d <_m a \Rightarrow \neg(a * c = b * d)$.

Lemma *ForeignRed*$^+$: $\forall a, b$: *term*. $\forall p, q$: *poly*.
foreign$(a \stackrel{>}{+} p, b \stackrel{>}{+} q) \Rightarrow$ *Spoly*$(a \stackrel{>}{+} p, b \stackrel{>}{+} q) \to^+_{[b \stackrel{>}{+} q]} ((-1).q) * (a \stackrel{>}{+} p)$.

The first lemma follows from the facts that a product $p * q$ always reduces by $[q]$ into a smaller (with respect to the order) product $p' * q$ and that the only product $p * q$ that is irreducible by $[q]$ is 0. The second lemma is proved by contradiction showing that if $a * c = b * d$, then $b * d <_m a\hat{}b$ and $b * d$ is divisible by a and b. The third lemma needs more work. We first have to show that *Spoly*$(a \stackrel{>}{+} p, b \stackrel{>}{+} q) = b.p - a.q$. Then from the second lemma we get that the terms of $b.p$ and of $a.q$ are not collapsing together in the subtraction, that is, all the terms of $b.p$ are in $b.p - a.q$. Each term of $b.p$ is divisible by b, so it can be reduced by $b \stackrel{>}{+} q$. The third lemma is proved by reducing all the terms of $b.p$ in $b.p - a.q$ starting from the smallest ones to avoid interference. By combining the third lemma and the first one, we get the theorem *ForeignRed**.

6.2. SECOND CRITERION

The second criterion can be expressed by a weakening of the predicate *SpolyP*:

Inductive *SP*$[S :$ (*list poly*)$]$: (*list poly*) $\to$ (*list poly*) $\to$ *Prop* :=
$\forall p, q$: *poly*. *Spoly*$(p, q) \to^*_S 0 \Rightarrow SP_S(p, q)$
| $\forall a, b, c$: *term*. $\forall p, q, r$: *poly*. $b \stackrel{>}{+} q$ *in* $S \Rightarrow$ *divP*$(a\hat{}c, b) \Rightarrow$
$SP_S(a \stackrel{>}{+} p, b \stackrel{>}{+} q) \Rightarrow SP_S(b \stackrel{>}{+} q, c \stackrel{>}{+} r) \Rightarrow SP_S(a \stackrel{>}{+} p, c \stackrel{>}{+} r)$.

Inductive $SpolyP_1[S: (list\ poly)]: Prop :=$
$(\forall p, q: poly.\ p\ in\ S \Rightarrow q\ in\ S \Rightarrow SP_S(p, q)) \Rightarrow SpolyP_1(S).$

If $SpolyP_1$ holds, it means that if a spolynomial may not reduce to zero, there exist some intermediate spolynomials that reduce all to zero. In the proof in Appendix B, we are using the fact that spolynomials reduce to zero under the condition that the relation is confluent for polynomials smaller than a given p. An induction proof shows that this condition and the fact that $SpolyP_1$ holds are sufficient to get the confluence for p. We thus derive the following theorem:

Theorem $Spoly_1ImpConf: \forall S: (list\ poly).\ SpolyP_1(S) \Rightarrow Confluent(S).$

6.3. THE OPTIMIZED ALGORITHM

The two criteria we have just proved can be used to optimize our algorithm. As we are going to reduce the number of generated spolynomials, only the function *SpolyL* has to be refined:

Inductive $cpRes: Set :=$
$\quad Keep: (list\ poly) \rightarrow cpRes$
$\mid\ DontKeep: (list\ poly) \rightarrow cpRes.$

Fixpoint $SpolyL\ [p: poly; S, T: (list\ poly)]: (list\ poly) :=$
Cases T **of**
$\quad [] \Longrightarrow S$
$\mid [q|T_1] \Longrightarrow$ **Cases** $Slice(p, q, T_1)$ **of**
$\quad\quad Keep(T_2) \Longrightarrow [Spoly(p, q)|SpolyL(p, S, T_2)]$
$\quad \mid DontKeep(T_2) \Longrightarrow SpolyL(p, S, T_2)$
$\quad$ **end**
end.

Instead of automatically adding the spolynomial of p and q, we make use of an auxiliary function *Slice* to decide whether we should keep it. Furthermore, this function can remove some elements from T_1 by returning a reduced list T_2. It is in the definition of the function *Slice* that we make use of the two criteria:

Fixpoint $Slice\ [p, q: poly; S: (list\ poly)]: cpRes :=$
Cases S **of**
$\quad [] \Longrightarrow$ **Cases** $foreignDec(p, q)$ **of**
$\quad\quad \{foreign(p, q)\} \Longrightarrow DontKeep([])$
$\quad \mid \{\neg foreign(p, q)\} \Longrightarrow Keep([])$
$\quad$ **end**
$\mid [r|S_1] \Longrightarrow$ **Cases** $divPDec(p\hat{}q, r)$ **of**
$\quad\quad \{divP(p\hat{}q, r)\} \Longrightarrow DontKeep(S)$
$\quad \mid \{\neg divP(p\hat{}q, r)\} \Longrightarrow$ **Cases** $divPDec(p\hat{}r, q)$ **of**
$\quad\quad\quad \{divP(p\hat{}r, q)\} \Longrightarrow Slice(p, q, S_1)$

```
        | {¬divP(p^r, q)} ⟹ addRes(r, Slice(p, q, S1))
        end
      end
  end.
```

where the definition *addRes* is

```
Fixpoint addRes [p : poly; res : cpRes] : cpRes :=
  Cases res of
        Keep(L) ⟹ Keep([p|L])
    | DontKeep(L) ⟹ DontKeep([p|L])
  end.
```

and the functions ^, *divP* and *divPDec* on polynomials are defined as the application of the corresponding functions on terms to the leading terms of the polynomials with the appropriate extension for the null polynomial.

We have proved this new version of the algorithm to be correct. Note that, from the point of view of program certification, the correctness of the optimized version is the really interesting part. The two criteria give us a way to avoid computation, telling us in advance that a given spolynomial will reduce to zero. If the application of the first criterion is direct, the application of the second one is more tricky and error-prone. A program that is too aggressive and discards wrongly some spolynomials could be very difficult to spot. The generation of spolynomials being heavily redundant, this program could still return correct results. One main benefit of our approach is to bridge the gap between the mathematical properties that justify optimizations and their use in the implementation. All the steps are formally justified. We are capable of safely deriving new properties that may be more suitable for establishing the correctness. While actually proving the correctness, we also rely on the prover to do all the necessary bookkeeping to ensure that we cover all the possible executions.

6.4. REDUCED BASIS

A program that computes Gröbner bases usually returns reduced bases, that is, bases where each polynomial is irreducible by the others. We define a nonoptimal algorithm to compute the reduced basis:

```
Fixpoint Redf [S, T : (list poly)] : (list poly) :=
  Cases S of
        [] ⟹ []
    | [p|S1] ⟹ let z = reducef_{S1+T}(p) in
        Cases zeroPDec(z) of
          {zeroP(z)} ⟹ Redf(S1, T)
```

$|\{\neg zeroP(z)\} \Longrightarrow [z|Redf(S_1, [z|T])]$
end
end.

Definition *Red*: (*list poly*) → (*list poly*) := λ*S*: (*list poly*). *Redf*(*S*, []).

We prove that the application of *Red* does not change the ideal:

Theorem *RedStable*: ∀*S*: (*list poly*). *Stable*(*S*, *Red*(*S*)).

Moreover, it preserves the property of being a Gröbner basis:

Theorem *RedGröbner*: ∀*S*: (*list poly*). *Gröbner*(*S*) ⇒ *Gröbner*(*Red*(*S*)).

By combining the functions *Buch* and *Red* we get our final function *BuchRed* that computes reduced Gröbner bases:

Definition *BuchRed*: (*list poly*) → (*list poly*) :=
λ*S*: (*list poly*). *Red*(*Buch*(*S*)).

We have limited ourselves to proving that *BuchRed* returns a Gröbner basis. Proving that what we obtain is a reduced basis is feasible but would require more work.

6.5. EXTRACTING THE ALGORITHM

So far our development is abstracted from the level of terms. Before extracting the algorithm, the last exercise is to instantiate terms as pairs of a coefficient and a monomial. For this we use a contribution by Loïc Pottier [23] that constructs monomials as lists of integers of fixed size. In doing so, our development becomes parameterized by a set of coefficients, which is supposed to be a field, and by a relation on monomials, which is supposed to be a monomial ordering. Appendix C gives the fully generalized version of the theorem *BuchGröbner*.

What the extraction mechanism [20] does when we ask to extract the function *BuchRed* is to take all the definitions that the function depends on and extract only the computational part. The result is a generic code that can be applied to an arbitrary field and an arbitrary monomial ordering. The self-contained extracted Ocaml program is 660 lines long. The examples below use an instantiation of the algorithm with polynomials of dimension 6 over $\mathbb{Q}$ with the usual graded inverse lexicographic ordering (`a > b > c > d > e > f`).* The interface to the code is composed of five 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 a polynomial by an integer;
3. `plus: poly -> poly -> poly` adds two polynomials;
4. `mult: poly -> poly -> poly` multiplies two polynomials;

* We are using the implementation of exact rational arithmetic described in [18].

5. `buchred: poly list -> poly list` computes the reduced Gröbner basis of a list of polynomials.

We also write a pretty-printer in Ocaml to make the output of computation more readable. We present below an interactive session with the Ocaml top-level. 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 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 +bc +ad +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
```

The computation of the reduced Gröbner basis returns the result:

```
# buchred [r3;r2;r1;r0];;
- : poly list = [a +b +c +d ;
                 b^2 +2bd +d^2 ;
                 bc^2 +c^2d -bd^2 -d^3 ;
                 bcd^2 +c^2d^2 -bd^3 +cd^3 -d^4 -1 ;
                 c^3d^2 +c^2d^3 -c -d ;
                 bd^4 +d^5 -b -d ;
                 c^2d^4 +bc -bd +cd -2d^2 ]
```

The answer is immediate on a DecAlpha 533 Mhz while the computation for $n = 5$ takes two seconds and the one for $n = 6$ thirty minutes. In comparison, MapleVr5 returns the result after five seconds for $n = 5$ and after fifteen minutes for $n = 6$.

7. Some Comments on the Proof Development

We hope that the outline of the development presented in the previous sections shows how naturally definitions and properties can be expressed in Coq. This is not specific to Coq but to any proof assistant based on higher-order logic. We believe that the same definitions and proof steps could be used to get the proofs of correctness in any theorem prover like Nuprl [7], HOL [12], Isabelle [22], and PVS [25].

7.1. COMPARISON WITH THE ORIGINAL TEXTBOOK

It is interesting to contrast what we have presented with our initial reference [11]. First of all, the chapter on Gröbner bases takes for granted all the basic properties of polynomials. In a prover, these properties need to be formalized. Second, if the general structure of the development follows the steps of what is presented in [11] closely, proof arguments often differ. The two central proofs given in Appendixes A and B are good examples of the differences that can occur. Often in textbooks inductive proofs include sentences such as "without loss of generality we can suppose that ...." In a prover, the induction principle has to be stated explicitly. In Appendix A, we replace the argument given in [11] by a simple induction on $Cb_S(p)$.

Proofs in Coq are done by using tactics. A tactic takes a goal and reduces it into a (possibly empty) list of simpler subgoals. The initial goal is the proposition to be proved, and the proof is finished when no subgoals are left. Appendix D gives the Coq script that we had to write to follow the steps of Appendix A. In this script we mainly use the tactic `Elim` to unfold inductive definitions and the tactic `Apply thm with v := value` to explicitly apply theorems that require witnesses. Since proof scripts using tactics are procedural, they are difficult to understand without replaying them. Using a prover with a declarative proof language à la Mizar [24] would make proof scripts more readable.

The second criterion that we have defined in Section 6.2 is a characteristic example where we have made use of the expressiveness of the language. In [11], the property of discarding the spolynomials is proved for a single intermediate polynomial. Using an inductive definition captures the constraint that this discarding process should be done in a well-founded way.

Finally, the programs we have described are rather different from what is presented in [11]. They are given in a functional style rather than an imperative one. Only two programs really depart from what is presented in [11]:

- The optimized version of the final algorithm immediately computes the spolynomials and applies the criteria rather than doing it when it is needed.
- The computation of reduced bases makes use of the fact that the argument is already a Gröbner basis to reduce it.

These modifications have been done in order to ease the proving process.

7.2. FORMALIZATION IN COQ

The entire development took us one year and a half to complete. It consists of 14,000 lines. Of these, 3000 lines are used to build monomials and terms and another 3000 lines to build polynomials. The construction of monomials includes the 900 lines developed by Loïc Pottier [23] that define the monomials as finite lists of numbers and gives a nonconstructive proof of Dickson's lemma. The central part of the development that defines the reduction and proves its main properties takes 5000 lines. An extra 1000 lines are dedicated to the proofs of the criteria. Finally, the part that corresponds to the definition of the algorithm and its proof of correctness is only 1000 lines.

Three main problems have slowed our development. First of all, because terms are abstracted from, each of our theorems needs to be fully quantified over the term structure in order to allow later instantiation. Coq provides a partial solution with its mechanism of local variables to avoid having to explicitly quantify. Unfortunately, this mechanism works inside a section that cannot exceed a single file, so it is not directly applicable for large developments. Then, in order to get all the theorems for a given instantiation, we need to operate on each of them individually, while instead we would like to globally perform the instantiation. These are well-known problems of modularity for which solutions have been proposed and implemented in other provers (see, for example, [10]). Clearly, modularity is a must if we aim at large proof developments.

Second, using an explicit equality as we do for terms and polynomials makes proofs harder in Coq because we do not get for free the possibility of replacing equals by equals. The 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 if we could work in a prover that supports such a capability.

Finally, Coq is equipped with little automation with respect to other provers such as Isabelle and PVS. We mostly use the tactic `Auto` that takes a list of theorems and checks if the goal is a consequence of the assumptions and the given list, using the modus ponens only and no logic variables. To overcome this lack of automation, we have taken special care to define an appropriate set of theorems such that most of the trivial subgoals are solved directly by `Auto`. As a consequence, if we take the number of theorems and the number of lines of our development, we get a ratio of 17 lines per theorem, which is reasonable. Still, we believe that our development could largely benefit from more automation, especially in the construction of terms and polynomials.

8. Related Work

Analytica [1] and more recently Theorema [4] propose an extension of the computer algebra system Mathematica [29] with a proving component. The examples they present are promising, but their proof engines seem to need further develop-

ment 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 attempts was the Automath project [19]. The current largest attempt is the Mizar project [24]. Some recent efforts include Jackson's work on computational algebra [16] and Harrison's work on real analysis [13]. The focus of these works is mostly on formalizing mathematics inside a prover, so they give very little account of algorithmic aspects. An exception is Schwarzweller's work [26], where a simple algebraic algorithm is proved correct using a verification condition generator and the prover Mizar.

Recently there have been other works on formalizing Gröbner bases and Buchberger's algorithm. Coquand and Persson mechanize a constructive proof of Dickson's lemma using open induction [8]. They plan to use this proof to get a fully constructive proof of the existence of Gröbner bases from which they can extract an algorithm. Also, Perez Vega and Werner have been working on a formalization of Buchberger's algorithm in Coq using a representation of polynomials as bags of terms [28].

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, [5]). In [14], there is a discussion on some of the limitations of this approach. In fact it is this work that has motivated our initial interest in proving Buchberger's algorithm.

9. Conclusion and Future Work

If we do not succeed in effectively mixing proving and computing since our computations are done in Ocaml, our contribution is to propose a common base where it is possible to reason about polynomials *and* develop a machine-checked and relatively efficient implementation that computes Gröbner bases. The 14,000 lines we had to write in order to prove 660 lines of code represent an important effort. Obviously, since algorithms may rely on very deep mathematical properties, there is not a canonical ratio between the proving part and the computing one. Still, we are aware that, in order to give evidence that this approach is practical for developing certified algorithms for computer algebra systems, we need to reduce the ratio to a more reasonable figure. A first step is clearly to increase the basic knowledge of provers. We have built our proof nearly from scratch. In particular, the construction of polynomials takes more than a third of the development. More automation and a better support to structure the development are also mandatory.

This initial experiment can be extended in several ways. First of all, it would be very interesting to see how the same proof looks in other theorem-proving systems. It would give a more accurate view of what current theorem-proving technology can achieve on this particular problem. We also want 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 [9], 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. 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 reason about. There have been some proposals for doing this inside a theorem prover (see, for example, [6]) but, as far as we know, there has been no concrete attempt to show how practical these solutions are.

Acknowledgments

We thank Bruno Buchberger who suggested his algorithm as a possible challenge at a Calculemus meeting in Rome in 1996, Loïc Pottier for his support and advice, and Monica Nesi and the referees for their remarks on the manuscript.

Appendix

A. Confluence Implies Gröbner

- We take an arbitrary polynomial p such that $Cb_S(p)$. We want to prove that $p \to_S^* 0$ under the hypothesis that the reduction is confluent.
- We proceed by induction on $Cb_S(p)$.
- In the base case, we have $p = 0$, so the property holds.
- In the induction case, we have $p = (a.r) + q$ with r *in* S and the induction hypothesis that $q \to_S^* 0$.
- We have $p - q = (a.r)$, with r *in* S. It implies that $p - q \to_S^+ 0$.
- By applying the lemma *Red*$^+$*Minus0*, we deduce that there exists an r such that $p \to_S^+ r$ and $q \to_S^+ r$.
- We know that the reduction is confluent and that q reduces to zero. It implies that r reduces to 0. So we get $p \to_S^* 0$. □

B. Spoly Implies Confluence

- The main hypothesis is *SpolyP(S)*.
- We want to prove that $\forall p, q, r\colon poly.\ p \to_S^* q \Rightarrow p \to_S^* r \Rightarrow q = r$.
- We prove it by induction on p using the theorem $<$*Ind*.
- The induction hypothesis is

$$\forall q\colon poly.\ q < p \Rightarrow \forall r, s\colon poly.\ q \to_S^* r \Rightarrow q \to_S^* s \Rightarrow r = s.$$

- We take two arbitrary reductions of p: $p \to_S^* r$ and $p \to_S^* s$ and prove that $r = s$.
- If p is irreducible, the property clearly holds $r = p = s$.
- Otherwise, we consider p_1 and p_2 such that $p \to_S p_1 \to_S^* r$ and $p \to_S p_2 \to_S^* s$.

- Because $p_1 < p$ and $p_2 < p$, it is 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.
- As p is reducible and in particular is not null, we have $p = t \hat{+} q$ for some term t and some polynomial q.
- 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 $p \rightarrow_S t \hat{+} q_1$ and $p \rightarrow_S t \hat{+} q_2$.
     - Since $q < p$, $q \rightarrow_S q_1$, 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 $t \hat{+} q_1 \rightarrow_S^+ t \hat{+} \mathit{reducef}_S(q)$ and $t \hat{+} q_2 \rightarrow_S^+ t \hat{+} \mathit{reducef}_S(q)$.
     - It is then sufficient to take $p_3 = \mathit{reducef}_S(t \hat{+} \mathit{reducef}_S(q))$.
  2. Suppose $p \rightarrow_S q - (t/t_1).q_1$ and $p \rightarrow_S t \hat{+} q_2$.
     - Since $q \rightarrow_S q_2$, by applying the lemma *RedDistMinus*, there exists a polynomial q_3 such that $q - (t/t_1).q_1 \rightarrow_S^+ q_3$ and $q_2 - (t/t_1).q_1 \rightarrow_S^+ q_3$.
     - We have by definition $t \hat{+} q_2 \rightarrow_S q_2 - (t/t_1).q_1$, so $t \hat{+} q_2 \rightarrow_S^+ q_3$.
     - It is then sufficient to take $p_3 = \mathit{reducef}_S(q_3)$.
  3. Suppose $p \rightarrow_S t \hat{+} q_1$ and $p \rightarrow_S q - (t/t_2).q_2$.
     - This case is just the symmetric of case 2, so the property holds.
  4. Suppose $p \rightarrow_S q - (t/t_1).q_1 = p_1$ and $p \rightarrow_S q - (t/t_2).q_2 = p_2$.
     - We have $p_1 - p_2 = (t/t_2).q_2 - (t/t_1).q_1$.
     - t_1 and t_2 divide t. We deduce that there exists t_3 such that $t = t_3.(t_2 \hat{} t_1)$.
     - So we get $p_1 - p_2 = t_3.\mathit{Spoly}(t_2 \hat{+} q_2, t_1 \hat{+} q_1)$.
     - Using the main hypothesis, we have $\mathit{Spoly}(t_2 \hat{+} q_2, t_1 \hat{+} q_1) \rightarrow_S^* 0$, 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 $p_3 = \mathit{reducef}_S(p_4)$.

- In all four cases, we are able to find such a polynomial p_3, so the property holds.

C. Fully Generalized Version of the Theorem *BuchGröbner*

Theorem *BuchGröbner*:
$\forall A\colon Set.$
$\forall =\colon A \Rightarrow A \Rightarrow Prop.$
$\forall eqRef\colon (reflexive\ A\ =).$
$\forall eqSym\colon (symmetric\ A\ =).$
$\forall eqTrans\colon (transitive\ A\ =).$
$\forall eqDec\colon \forall a, b\colon A.\ \{a = b\} + \{\neg(a = b)\}.$
$\forall 0\colon A.$
$\forall 1\colon A.$
$\forall 1Diff0\colon \neg(1 = 0).$
$\forall +\colon A \Rightarrow A \Rightarrow A.$
$\forall plusAssoc\colon \forall a, b, c\colon A.\ (a + (b + c)) = ((a + b) + c).$
$\forall plusCom\colon \forall a, b\colon A.\ (a + b) = (b + a).$
$\forall plusEqComp\colon \forall a, b, c, d\colon A.\ a = c \Rightarrow b = d \Rightarrow (a + b) = (c + d).$
$\forall plus0\colon \forall a\colon A.\ a = (a + 0).$
$\forall -\colon A \Rightarrow A.$
$\forall invPlus\colon \forall a\colon A.\ 0 = (a + (-a)).$
$\forall invEqComp\colon \forall a, b\colon A.\ a = b \Rightarrow (-a) = (-b).$
$\forall -\colon A \Rightarrow A \Rightarrow A.$
$\forall minusDef\colon \forall a, b\colon A.\ (a - b) = (a + (-b)).$
$\forall *\colon A \Rightarrow A \Rightarrow A.$
$\forall mult0L\colon \forall a\colon A.\ (0 * a) = 0.$
$\forall mult1L\colon \forall a\colon A.\ (1 * a) = a.$
$\forall multEqComp\colon \forall a, b, c, d\colon A.\ a = c \Rightarrow b = d \Rightarrow (a * b) = (c * d).$
$\forall multAssoc\colon \forall a, b, c\colon A.\ (a * (b * c)) = ((a * b) * c).$
$\forall multCom\colon \forall a, b\colon A.\ (a * b) = (b * a).$
$\forall multDistL\colon \forall a, b, c\colon A.\ ((c * a) + (c * b)) = (c * (a + b)).$
$\forall /\colon A \Rightarrow \forall b\colon A.\ \neg(b = 0) \Rightarrow A.$
$\forall divIsMult\colon \forall a, b\colon A.\ \forall nZb\colon \neg(b = 0).\ a = ((a/b) * b).$
$\forall divEqComp\colon \forall a, b, c, d\colon A.\ \forall nZb\colon \neg(b = 0).\ \forall nZd\colon \neg(d = 0).$
$\quad a = c \Rightarrow b = d \Rightarrow (a/b) = (c/d)$
$\forall divMultCompR\colon \forall a, b, c\colon A.\ \forall nZc\colon \neg(c = 0).\ ((a * b)/c) = ((a/c) * b).$
$\forall divIrr\colon \forall a, b\colon A.\ \forall nZ_1, nZ_2\colon \neg(b = 0).\ a/_{nZ_1}b == a/_{nZ_2}b.$
$\forall n\colon nat.$
$\forall <\colon (mon\ n) \Rightarrow (mon\ n) \Rightarrow Prop.$
$\forall 1Min\colon \forall a\colon (mon\ n).\ \neg(a < (M1\ n)).$
$\forall ltNonRefl\colon \forall x\colon (mon\ n).\ \neg(x < x).$
$\forall ltTrans\colon (transitive\ (mon\ n)\ <).$
$\forall ltDec\colon \forall x, y\colon (mon\ n).\ \{x < y\} + \{y < x\} + \{x = y\}.$
$\forall ltWf\colon (wellFounded\ (mon\ n)\ <).$
$\forall ltPlusR\colon \forall a, b, c\colon (mon\ n).\ a < b \Rightarrow (a * c) < (b * c).$
$\forall ltPlusL\colon \forall a, b, c\colon (mon\ n).\ a < b \Rightarrow (c * a) < (c * b).$

$\forall P$: (*list poly*). (*Gröbner* (*Buch P*)).

D. The Proof Script of Confluence Implies Gröbner

```
Theorem ConfluentReduce_imp_Grobner:
  (Q:(list poly)) (ConfluentReduce Q) -> (Grobner Q).

Intros Q H'; Elim H'.
Intros H'0.
Apply Grobner0; Auto.
Intros p q H'1; Generalize q; Clear q; Elim H'1.
Intros q H'2.
Rewrite (pO_reducestar Q q); Auto.
Intros a p0 q s H'2 H'3 H'4 H'5 H'6 q0 H'7.
Cut (canonical q);
  [Intros Op0 | Apply inPolySet_imp_canonical with L := Q];
  Auto.
Cut (canonical p0);
  [Intros Op2 | Apply CombLinear_canonical with Q := Q];
  Auto.
Cut (canonical s); [Intros Op1 | Idtac].
Cut (canonical q0); [Intros Op2b | Idtac]; Auto.
LApply (reducestar_in_pO Q a q);
 [Intros H'11; LApply H'11; [Intros H'12 | Idtac]
 | Idtac]; Auto.
Elim (red_minus_zero_reduce Q s p0);
 [Intros r1 E; Elim E; Intros H'15 H'16; Clear E
 | Idtac
 | Idtac
 | Idtac]; Auto.
Elim (reduce0_reducestar Q r1); [Intros t E | Idtac]; Auto.
LApply (H'0 s); [Intros H'11;Inversion H'11 | Idtac]; Auto.
Apply H; Auto.
Apply reducestar_eqp_com with p := s q := t; Auto.
Apply reducestar_trans with y := r1; Auto.
Apply H'5; Auto.
Apply (reducestar_trans Q) with y := r1; Auto.
Apply canonical_reduceplus with Q := Q p := s; Auto.
Apply reduceplus_eqp_com with p := (multm_lm a q)
                              q := pO; Auto.
Apply eqp_sym; Apply eqp_trans with y :=
  (minuspf (pluspf (multm_lm a q) p0) p0); Auto.
Apply eqp_trans with y :=
  (pluspf (pluspf (multm_lm a q) p0)
```

```
          (multm_lm (invTerm T1) p0)); Auto.
Apply eqp_trans with y :=
  (pluspf (multm_lm a q)
          (pluspf p0 (multm_lm (invTerm T1) p0)));
 Auto.
Apply eqp_trans with y := (pluspf (multm_lm a q) p0); Auto.
Apply canonical_reduceplus with Q := Q p := s; Auto.
Apply eqp_imp_canonical with p :=
  (pluspf (multm_lm a q) p0); Auto.
Apply eqp_sym; Auto.
Qed.
```

References

1. Bauer, A., Clarke, E. and Zhao, X.: Analytica – an experiment in combining theorem proving and symbolic computation, *J. Automated Reasoning* **21**(3) (1998), 295–325.
2. Bertot, Y. and Bertot, J.: CtCoq: A system presentation, in *Automated Deduction – CADE-13*, Lecture Notes in Artif. Intell. 1104, Springer-Verlag, 1996.
3. Buchberger, B.: Introduction to Gröbner bases, in B. Buchberger and F. Winkler (eds.), *Gröbner Bases and Applications*, Cambridge University Press, 1998, pp. 3–31.
4. Buchberger, B., Jebelean, T., Kriftner, F., Marin, M., Tomuta, E. and Vasaru, D.: A survey on the theorema project, in *International Symposium on Symbolic and Algebraic Computation (ISSAC'97)*, ACM, 1997.
5. Calmet, J. and Homann, K.: 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 Appl. Logic, Springer-Verlag, 1996, pp. 133–146.
6. Constable, R. L.: Expressing computational complexity in constructive type theory, in *International Workshop on Logic and Computational Complexity*, Lecture Notes in Artif. Intell. 960, Springer-Verlag, July 1994.
7. Constable, R. L., Allen, S. F., Bromley, H. M., Cleaveland, W. R., Cremer, J. F., Harper, R. W., Howe, D. J., Knoblock, T. B., Mendler, N. P., Panangaden, P., Sasaki, J. T. and Smith, S. F.: *Implementing Mathematics with Nuprl Proof Development System*, Prentice-Hall, 1986.
8. Coquand, T. and Persson, H.: Gröbner bases and type theory, in T. Altenkirch, W. Naraschewski, and B. Reus (eds.), *Types for Proofs and Programs*, Lecture Notes in Comput. Sci. 1657, Springer-Verlag, 1999.
9. Coscoy, Y., Kahn, G. and Théry, L.: Extracting text from proofs, in *Typed Lambda Calculus and Its Applications*, Lecture Notes in Comput. Sci. 902, Springer-Verlag, 1995, pp. 109–123.
10. Farmer, W. M., Guttman, J. D. and Thayer, F. J.: Little theories, in D. Kapur (ed.), *Automated Deduction – CADE-11*, Lecture Notes in Comput. Sci. 607, Springer-Verlag, 1992, pp. 567–581.
11. Geddes, K. O., Czapor, S. R. and Labahn, G.: *Algorithms for Computer Algebra*, Kluwer Acad. Publ., 1992.
12. Gordon, M. and Melham, T.: *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, Cambridge Univ. Press, 1993.
13. Harrison, J. R.: *Theorem Proving with the Real Numbers*, Springer-Verlag, 1998.
14. Harrison, J. R. and Théry, L.: A skeptic's approach to combining HOL and Maple, *J. Automated Reasoning* **21**(3) (1998), 295–325.

15. Huet, G., Kahn, G. and Paulin-Mohring, C.: The Coq proof assistant: A tutorial: Version 6.1, Technical Report 204, INRIA, 1997.
16. Jackson, P. B.: Enhancing the Nuprl proof development system and applying it to computational abstract algebra, Technical Report TR95-1509, Cornell University, 1995.
17. Knuth, D. E. and Bendix, P. B: Simple word problems in universal algebras, in J. Leech (ed.), *Computational Problems in Abstract Algebras*, Pergamon Press, 1970.
18. Ménissier-Morain, V.: The CAML numbers reference manual, Technical Report 141, INRIA, 1992.
19. Nederpelt, R. P., Geuvers, J. H. and De Vrijer, R. C. (eds.): *Selected Papers on Automath*, North-Holland, 1994.
20. Paulin-Mohring, C. and Werner, B.: Synthesis of ML programs in the system Coq, *J. Symbolic Comput.* **15**(5–6) (1993), 607–640.
21. Paulson, L. C.: Constructing recursion operators in intuitionistic type theory, *J. Symbolic Comput.* **2**(4) (1986), 325–355.
22. Paulson, L. C.: *Isabelle: A Generic Theorem Prover*, Lecture Notes in Comput. Sci. 828, Springer-Verlag, 1994.
23. Pottier, L.: Dickson's lemma, Available at `ftp://ftp-sop.inria.fr/lemme/Loic.Pottier/MON/`, 1996.
24. Rudnicki, P.: An overview of the MIZAR projet, in *Workshop on Types and Proofs for Programs*, Available by `ftp` at `pub/cs-reports/baastad.92/proc.ps.Z` on `ftp.cs.chalmers.se`, 1992.
25. Rushby, J. M., Shankar, N. and Srivas, M.: PVS: Combining specification, proof checking, and model checking, in *CAV '96*, Lecture Notes in Comput. Sci. 1102, Springer-Verlag, July 1996.
26. Schwarzweller, C.: Mizar verification of generic algebraic algorithms, Ph.D. Thesis, Wilhelm-Schickard Institute for Computer Science, University of Tübingen, 1997.
27. Théry, L.: A certified version of Buchberger's algorithm, in *Automated Deduction – CADE-15*, Lecture Notes in Artif. Intell. 1421, Springer-Verlag, 1998, pp. 349–364.
28. Vega, G. P. and Werner, B.: Personal communication, 1998.
29. Wolfram, S.: *Mathematica: A System for Doing Mathematics by Computer*, Addison-Wesley, 1988.