View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by idUS. Depósito de Investigación Universidad de Sevilla

A verified COMMON LISP implementation of Buchberger's algorithm in ACL2

Inmaculada Medina-Bulo [a], Francisco Palomo-Lozano [a], José-Luis Ruiz-Reina [b]

[a] *Escuela Superior de Ingeniería, Calle Chile s/n 11003 Cádiz, Spain*
[b] *ETS de Ingeniería Informática, Avda. Reina Mercedes s/n 41012 Sevilla, Spain*

Keywords:
Formal verification
ACL2
Computer Algebra
Buchberger's algorithm

A B S T R A C T

In this article, we present the formal verification of a COMMON LISP implementation of Buchberger's algorithm for computing Gröbner bases of polynomial ideals. This work is carried out in ACL2, a system which provides an integrated environment where programming (in a pure functional subset of COMMON LISP) and formal verification of programs, with the assistance of a theorem prover, are possible. Our implementation is written in a real programming language and it is directly executable within the ACL2 system or any compliant COMMON LISP system. We provide here snippets of real verified code, discuss the formalization details in depth, and present quantitative data about the proof effort.

1. Introduction

Formal verification of a program consists in proving, using formal methods, that the program is correct with respect to a given formal specification of its intended behavior. Usually, this is achieved by reasoning about the program in a formal logic, assisted by a mechanical theorem prover. Formal verification dramatically increases our confidence in programs and provides us with a deeper understanding of them.

Computer Algebra Systems (CAS) are an interesting target for formal verification. First, formally verified implementations of symbolic computation algorithms would allow us to build more reliable and accurate components for these systems. Second, since the underlying mathematical theories in a CAS are usually much richer than in other software systems, the formal verification of these algorithms needs the development of formalizations of important mathematical theories, interesting on their own.

Buchberger's algorithm for computing Gröbner bases of multivariate polynomial ideals (Buchberger, 1970, 2006) has become one of the most important algorithms in Computer Algebra. Currently, this algorithm is available in most CAS and its theory, implementation and numerous applications are widely documented in the literature, e.g., Buchberger and Winkler (1998); Geddes et al. (1992). Our aim here is to describe the formal verification of a naive COMMON LISP implementation of Buchberger's algorithm. The implementation and formal proofs have been carried out in the ACL2 system, see Kaufmann et al. (2000b), which consists of a programming language (an extension of a pure functional or *applicative* subset of COMMON LISP), a logic for stating and proving program properties, and a theorem prover supporting mechanized reasoning in the logic. ACL2 has proved most successful in the formal verification of digital computing systems and now we apply it to the verification of a key Computer Algebra algorithm, showing the feasibility of doing programming, proving and computing concurrently and reliably.

Implementations of Buchberger's algorithm are difficult to test. For example, even if the algorithm fails to generate some s-polynomials (a key concept, as will be seen later on), this error may pass inadvertently through many test cases. The importance of Buchberger's algorithm in Computer Algebra justifies on its own the effort of obtaining a formal correctness proof with a theorem prover, and this is one of the motivations for this work. This goal has already been achieved by L. Théry. See Théry (2001) where he gives a formal proof using the COQ system and explains how an executable implementation in the OCAML language is extracted from the algorithm defined in COQ. In contrast, in ACL2 we can reason directly about the LISP program implementing the algorithm, i.e., about the very program which is executed by the underlying LISP system. There is a price to pay: the logic of ACL2 is a fragment of first-order logic, less expressive than the logic of COQ, which is based on higher-order type theory. Nevertheless, the degree of automation of the ACL2 theorem prover is higher than in other systems with more expressive logics.

We show how it is possible to formalize all the theory underlying Buchberger's algorithm within the ACL2 logic, finally obtaining a formally verified implementation of the algorithm executable in any compliant COMMON LISP, a language widely used in the implementation of CAS systems.[2] We also think that it is interesting to compare our approach with Théry's work, pointing out the main differences with his COQ formalization and proof development, as well as with other approaches.

In the following, we first give an introduction to ACL2 in Section 2. Then, from Section 3–7, we describe different parts of the formal theory developed, leading to the implementation and formal verification of Buchberger's algorithm presented in Section 8. Finally, we conclude with some comments about the proof development and related work.

2. An introduction to the ACL2 system

ACL2 stands for "A Computational Logic for an Applicative Common Lisp". This system descends from the Boyer–Moore theorem prover, also known as NQTHM (Boyer and Moore, 1998). Roughly speaking, ACL2 has three components: a programming language, a logic and a theorem prover. As a programming language, it is an extension of a useful applicative subset of COMMON LISP (Steele Jr., 1990).[3] The ACL2 logic describes the programming language with a formal syntax, axioms and rules of inference allowing to formally state and prove properties about the programs implemented in the

[2] We do not use any non-compliant feature of ACL2 in executable function definitions. Of course, we use a few primitives, whose definitions can be copied verbatim and accepted by any compliant LISP.

[3] We will assume that the reader is familiar with this language.

language. The logic is supported by an inductive theorem prover that provides mechanized reasoning. Thus, the system constitutes an environment in which algorithms can be defined and executed, and their properties can be formally specified and proved with the assistance of an automated theorem prover.

Let us illustrate these features with an example. The following functions, `insert` and `isort`, implement the standard insertion-based algorithm for sorting lists in ACL2:

```
(defun insert (x l)
  (if (consp l)
      (if (<= x (first l))
          (cons x l)
        (cons (first l) (insert x (rest l))))
    (cons x l)))

(defun isort (l)
  (if (consp l)
      (insert (first l) (isort (rest l)))
    l))
```

As it can be seen in this example, ACL2 is very similar to COMMON LISP. In order to allow reasoning about the ACL2 functions as functions in the mathematical sense, no side-effect features (e.g., global variables or destructive updates) are permitted. Furthermore, ACL2 functions are first-order: functional arguments are not possible.

Once defined, the functions can be executed on explicit values, as in any LISP interpreter. For example, the expression `(isort '(21 14 3 7))` evaluates to `'(3 7 14 21)`. In fact, these functions can be used verbatim in any LISP system compliant with the COMMON LISP standard, yielding the same results.

But we can also state and prove formal properties about them, using the ACL2 logic. From the logical point of view, the above definitions introduce two axioms in the ACL2 logic, equating the terms `(insert x l)` and `(isort l)` to the terms given by the bodies of their respective definitions. Using the definitional axioms involved, and the primitive axioms and rules of inference of the ACL2 logic, it is possible, with the assistance of the theorem prover, to prove the following property about `isort`, which hopefully states the intended requirements of the implementation:

```
(defthm isort-correctness
  (and (perm l (isort l))
       (ordered (isort l))))
```

The ACL2 logic provides both a formal language to state properties and a precise notion of what a theorem is. Note that *the same language* is used for programming and for specification. In the above formula, which happens to be an ACL2 theorem, the primitive `and` implements the propositional conjunction connective and the variable `l` is assumed to be implicitly universally quantified. Furthermore, `perm` and `ordered` are ACL2 functions implementing the intended meanings of "being a permutation" and "being ordered", respectively. Both functions are just needed for specifying the sorting algorithm and reasoning about its properties, but there is nothing special about them: they are implemented in the same way as any other function. For example, the following is the definition of `perm` (where `remove-one` is an auxiliary function removing one occurrence of a given element from a list):

```
(defun perm (l m)
  (cond ((endp l)
         (endp m))
        ((member (first l) m)
         (perm (rest l) (remove-one (first l) m)))
        (t
         nil)))
```

The interest in proving theorems in ACL2 comes from the following fact: the applicative subset of COMMON LISP constitutes a model for the ACL2 logic. That is, the ACL2 axioms are basic true facts in every compliant COMMON LISP and the rules of inference are basic principles deriving true facts from true facts. Assuming this,[4] every theorem proved in ACL2 must be true in any compliant COMMON LISP, since a theorem is derived by applying a finite number of axioms and rules of inference. In our example, this means that *for every* concrete value of τ, the evaluation of `(isort` τ`)` will *always* return an object which is an ordered permutation of τ, in *any* compliant COMMON LISP. This argument increases our confidence in the above implementation of `isort`.

In the following, we will briefly describe the ACL2 logic and its theorem prover. To obtain more background on ACL2, see Kaufmann et al. (2000b) or the user's manual and tutorials in Kaufmann and Moore (2009).

2.1. The logic

The ACL2 logic is a fragment of first-order logic with equality describing an applicative subset of COMMON LISP functions acting on a set of objects known as the ACL2 *universe*. This universe is partitioned into five sets of objects: numbers, characters, strings, symbols and ordered pairs or *conses*. Essentially, these data types are the same as in COMMON LISP and they are represented in the same way. There are two special symbols, `t` and `nil`, denoting respectively "true" and "false" (although any object other than `nil` may serve as "true").

The syntax of *terms* in the ACL2 logic is that of COMMON LISP, and therefore it uses prefix notation. Roughly speaking, a term of the ACL2 logic is a constant, a variable symbol or the application of a k-ary function symbol to k terms. As exemplified by `isort-correctness` above, *formulas* are quantifier-free and their free variables can be regarded as being universally quantified.

The logic includes axioms for propositional logic (with propositional connectives `and`, `or`, `not`, `implies` and `iff`) and equality (the predicate `equal`), as well as axioms describing the behavior of a subset of primitive COMMON LISP functions on the above five data types. For example, there are axioms describing `cons`, `first` and `rest` (the constructor, the first element and the second element of an ordered pair, respectively).

Rules of inference include those for propositional logic, equality and instantiation of variables. The logic also provides a principle of *proof by induction* that allows to prove a conjecture by splitting it into cases and inductively assuming some instances of the conjecture that are smaller with respect to some well-founded measure.

In ACL2, the primitive notion of well-foundedness is given by a constructive representation of the ordinals up to ϵ_0 in terms of natural numbers and ordered pairs. Finite ordinals are represented by the corresponding ACL2 natural number, while non-natural ordinal numbers are represented by the list of the ordered pairs of coefficients and exponents of its Cantor normal form. For example, $\omega^2 + \omega \cdot 4 + 7$ is represented as `'((2 . 1) (1 . 4) . 7)`. The ACL2 primitive function `o-p` recognizes those ACL2 objects that represent ordinals, while the function `o<` defines the usual order between ordinals. This order is assumed to be well-founded on the set of ACL2 ordinals.

The ACL2 logic is not static, in the sense that it can be extended with new function symbols and new non-logical axioms. This is mainly accomplished through two *extension principles*. In order to avoid inconsistencies, a careful treatment of extensions is required. For example, every time a new definition is introduced, a definitional axiom is added: the above definition of `isort` introduces the axiom `(equal (isort l)` *body*`)` where *body* is the body of the `isort` definition. Assume for a moment that we could introduce the definition `(defun f (x) (+ 1 (f x)))`: this clearly would be in contradiction with the arithmetic theorem `(not (equal n (+ 1 n)))`, rendering the logic inconsistent.

By the *principle of definition* (via `defun`), new function definitions are admitted as axioms in the logic only if an ordinal measure of the function arguments can be found such that this measure strictly

[4] And the correctness of the hardware and software used, including the theorem prover.

decreases in each recursive call (if any), thus proving its termination. For example, the admission of `isort` in the logic is justified by a measure counting the number of "conses" of its argument. It can be shown that the definitional axiom introduced by a terminating function does not introduce inconsistencies. Note that ACL2 functions are *total*: that is, they return an ACL2 object for every input, even for inputs outside their intended domains.

In addition to the definitional principle, the *encapsulation principle* (via `encapsulate`) allows the user to introduce new function symbols constrained by axioms to have certain properties. To ensure consistency, local witness functions with the same properties have to be exhibited.[5] Within the scope of an `encapsulate`, properties stated with `defthm` need to be proved for the witnesses; outside, these theorems work as assumed axioms.

For example, the following encapsulation introduces a new unary function symbol `sel` constrained to denote a function that selects an element from every non-empty list:

```
(encapsulate
  (((sel *) => *))                  ; Signature list

  (local                            ; Local witness
    (defun sel (l)
      (first l)))

  (defthm sel-selects               ; Non-local property
    (implies (consp l)
             (member (sel l) l))))
```

The first part of every `encapsulate` describes the *signatures* of the function symbols introduced (in this case, `sel`, with one argument and one result). In this example, the local witness is the function `sel` which just takes the first element of a non-empty list and it is proved to satisfy the non-local property `sel-selects`. Outside the scope of the `encapsulate`, this property is the only knowledge we have about the function `sel`, as its definition is forgotten. With this assumed property, one could prove, for example, the following theorem:

```
(defthm sel-len
  (implies (consp l)
           (< (len (remove-one (sel l) l)) (len l))))
```

Although functions locally defined in an `encapsulate` cannot be executed, reasoning with their properties provides more generality. Encapsulated function symbols can be intuitively seen as second-order variables, representing functions verifying certain properties. A derived rule of inference, *functional instantiation*, allows us to reason in a "second-order" fashion: theorems about constrained functions can be instantiated by function symbols if they are known to have the assumed properties. For example, if we replace in the theorem `sel-len` the function symbol `sel` by another and prove that the latter corresponds to a function selecting an element for every non-empty list, then the functional instantiation rule guarantees that the resulting formula is a theorem, without having to replay the proof. See Kaufmann and Moore (2001) for details on functional instantiation.

2.2. The theorem prover

The ACL2 theorem prover is inspired by NqThm, adapted to the ACL2 logic and considerably enhanced. The main proof techniques used by ACL2 are simplification and induction.[6] Roughly speaking, when the prover tries to prove a conjecture, it simplifies the formula. If it obtains `t`, then

---

5 This guarantees the existence of, at least, one model of the extended theory.

6 Although there are other proof techniques like *destructor elimination*, *generalization*, *elimination of irrelevances* and *cross-fertilization*.

the conjecture is proved. Otherwise, it guesses an (often suitable) induction scheme, and recursively tries to prove the subgoals generated.

Simplification is a process combining term rewriting with some decision procedures (propositional logic, linear arithmetic, type-set reasoning, etc.). To rewrite a conjecture, the system uses axioms, definitions and theorems previously proved by the user, typically stored as conditional rewrite rules. In addition to simplification, one of the key points in the success of ACL2 and its predecessor is the use of sophisticated heuristics for discovering an induction scheme suitable for a proof by induction of a conjecture. This induction scheme is suggested by the recursive functions occurring in the formula.

For example, in the proof attempt of `(perm l (isort l))`, the system tries a proof by induction, since the conjecture cannot be simplified using the current definitions and rewrite rules. The following induction scheme is automatically generated, where `(Ψ l)` abbreviates the theorem to be proved:

```
(and (implies (and (not (endp l))
                   (not (member (first l) (isort l))))
              (Ψ l))                        ; Base case
     (implies (and (not (endp l))
                   (member (first l) (isort l))
                   (Ψ (rest l)))
              (Ψ l))                        ; Induction step
     (implies (endp l)
              (Ψ l)))                       ; Base case
```

This induction scheme is suggested by the recursive definition of the functions `isort` and `perm` appearing in the conjecture. It consists of two base cases and one induction step, where the induction hypothesis is `(Ψ (rest l))`, corresponding to the recursive call in both functions.

Although this induction scheme is suitable for proving the theorem, ACL2 fails to prove it in its first attempt. Inspecting the output of the failed proof, it turns out that the proof would succeed if the following lemma were known by the system:

```
(defthm remove-one-insert
  (equal (remove-one x (insert x l)) l))
```

When submitted by the user, this lemma is proved automatically by the system. Once proved, the lemma is stored as a rewrite rule. Subsequent proofs can use this rule to rewrite instances of `(remove-one x (insert x l))` to the corresponding instance of `l`. Now, a second proof attempt of the theorem `(perm l (isort l))` succeeds.

The theorem prover is automatic in the sense that once `defthm` is invoked, the user can no longer interact with the system. However, as we have just seen, in a deeper sense the system is interactive. As in the above example, non-trivial proofs are not found by the system in a first attempt very often and then the user has to guide the prover by adding lemmas and definitions, used in subsequent proofs as rules.[7] The lemmas needed to "program" the system in order to get the mechanical proof of a non-trivial result are discovered by the user from two sources: at a higher level, the user has a preconceived hand proof in mind and decomposes the proof task in lemmas according to it; at a lower level of detail, inspection of failed proofs suggests some necessary lemmas (see Kaufmann et al. (2000b) for a description of how to inspect failed proofs).

Thus, the role of the user is important: a typical proof effort consists in formalizing the problem in the logic (which could be difficult because of the limited expressiveness of the logic) and helping the prover to find a preconceived hand proof by means of a suitable set of rewrite rules. As a result of this interaction, the user obtains a number of files containing (mainly) definitions and theorems, called *books* in the ACL2 terminology. A book can be *certified*, which means that all its definitions are admissible and its theorems are proved, and used by other books.

[7] The user may also assist the prover by giving some explicit *hints* when submitting the conjecture, e.g., indicating the use of some specific lemma instance or providing an induction scheme.

In the following, we present our implementation of Buchberger's algorithm along with its formal verification in ACL2. We developed several certified ACL2 books and the collection of definitions and lemmas in them were obtained following a standard interaction with the system as explained above. The formal proofs developed in ACL2 are mainly adapted from Baader and Nipkow (1998, Chap. 8). As the whole development consists of roughly one thousand ACL2 theorems and function definitions, we will only scratch its surface presenting the main results and a sketch of how the pieces fit together, making emphasis on the formalization aspects. We will necessarily omit many details that, we expect, can be inferred from the context.

3. Polynomial rings

In order to introduce some terminology, let K be a field and $\{x_1, \ldots, x_k\}$ a set of k *variables*. A *monomial* is a finite power product of the form $c \cdot x_1^{a_1} \cdots x_k^{a_k}$, where $a_1, \ldots, a_k \in \mathbb{N}$ are its *exponents*, $c \in K$ is its *coefficient* and $x_1^{a_1} \cdots x_k^{a_k}$ is its *term*. A *polynomial* is a finite sum of monomials. $K[x_1, \ldots, x_k]$ denotes the set of polynomials in the variables $x_1, \ldots, x_k$ with coefficients in K.

In order to represent polynomials in ACL2, we have chosen a sparse, normalized and uniform representation. That is, having fixed the set of variables and an ordering between them, a canonical form can be associated to each term. We will represent a term as the list of its exponents; similarly a monomial will be represented as the list having its coefficient as the first element and its term as the rest. Furthermore, having fixed an arbitrary term ordering we can also associate a *canonical representation* to a given polynomial; in this canonical form all monomials are arranged in a strictly decreasing order, there are no null monomials and all of them have the same number of variables. In our case, the term ordering chosen is the *lexicographic ordering*, obtained by comparing lexicographically the successive exponents of the terms. We will represent a polynomial as the ordered list of the monomials in its canonical form. For example, with the lexicographic term ordering induced by $x > y$, the internal representation of the polynomial $2x^4y - xy^4 + 5x - 1/2$ in our formalization is the list `'((2 4 1) (-1 1 4) (5 1 0) (-1/2 0 0))`.

A computational theory of multivariate polynomials on a coefficient field was developed in Medina-Bulo et al. (2000, 2002). This ACL2 formalization includes common operations and fundamental properties establishing their ring structure. The aim was to develop a reusable library on polynomials. Although most of the theory deals with an arbitrary field via the encapsulation principle, we use polynomials over the field of rational numbers for our implementation of Buchberger's algorithm. This alleviates some proofs at the cost of some generality, as ACL2 can use its built-in rational linear arithmetic decision procedure. In any case, the general theory has to be eventually instantiated to obtain an executable algorithm.

In this library, the functions `monomialp` and `polynomialp` recognize ACL2 objects representing monomials and polynomials, respectively. The usual operations on polynomials are also defined, in particular, `+` (addition), `*` (multiplication), `-` (negation) and `|0|` (the zero polynomial).[8] For a detailed description of these functions, we refer the reader to Medina-Bulo et al. (2000, 2002). Essentially, the operations are implemented on top of an unnormalized representation of polynomials, applying a final normalization step that eliminates repeated terms and null monomials and sorts the monomials with respect to the term ordering.

Since our normalized representation requires that all the monomials in a polynomial have the same number of variables, `k`, some of the functions that will be introduced in the following have this `k` as a parameter. The most basic of these functions are `k-polynomialp` and `k-polynomialsp` which respectively recognize normalized polynomials and polynomial lists, with `k` variables and rational coefficients. In principle, we have to introduce this extra argument `k` in every definition that uses it to check that the monomials of the polynomials involved have a uniform number of variables. For the sake of readability, we have defined a macro `defun<k>`, similar to `defun`, but avoiding to explicitly mention the `k` parameter. This macro automatically generates the real `defun` event,

[8] In order to avoid name clashes with the corresponding operations on ACL2 numbers, we used ACL2 *packages*, similar to Lisp packages.

explicitly introducing the k argument. In particular, k-polynomialp and k-polynomialsp are introduced via defun<k>.

4. Polynomial ideals

Definition 1. Let R be a commutative ring. We say that $I \subseteq R$ is an *ideal* of R if it is closed under addition, and under the product by elements of R.

Definition 2. Let R be a commutative ring and $B \subseteq R$. We say that $\sum_{i=1}^{n} p_i \cdot f_i$ is a *linear combination* of B with coefficients in R if $p_i \in R$ and $f_i \in B$. The *ideal generated* by $B \subseteq R$, denoted as $\langle B \rangle$, is the set of all linear combinations of elements of B with coefficients in R. We say that B is a *basis* of $I \subseteq R$ if $I = \langle B \rangle$. An ideal is *finitely generated* if it has a finite basis.

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

Although ACL2 formulas are quantifier-free, it is possible to define functions whose body has an outermost quantifier by means of the predefined defun-sk macro. We use defun-sk to define the function in<>, which formalizes ideal membership (more precisely, we use a macro defun-sk<k> to avoid the explicit introduction of the k parameter, as explained in the preceding section):

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

The above construction automatically generates the following encapsulate, where a Skolem function in<>-witness, with arguments p and F, is introduced:[9]

```
(encapsulate
  (((in<>-witness * *) => *))
  ...
  (defun<k> in<> (p F)
    (let ((C (in<>-witness p F)))
      (and (k-polynomialsp C)
           (equal p (lin-comb C F)))))

  (defthm in<>-suff
    (implies (and (k-polynomialsp C)
                  (equal p (lin-comb C F)))
             (in<> p F))))
```

According to the definition above, if (in<> p F) then (in<>-witness p F) is a list of polynomial coefficients such that when linearly combined with the polynomials in F, p is obtained (i.e., (in<>-witness p F) is a witness of the membership of p to the ideal generated by F). Conversely, by the property in<>-suff, to establish (in<> p F) it suffices to explicitly provide such a witness list C of polynomial coefficients.

The following theorems establish that our definition of ideal in ACL2 meets the intended closure properties (event names delimited by | may include special characters but represent just convenient names for us which are meaningless for ACL2):

[9] We omit the definition of the local witness for in<>-witness. It is defined using defchoose, a way to define Skolem functions in ACL2. See Kaufmann and Moore (2001) for details on defun-sk and defchoose.

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

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

Both theorems are proved as explained above, using the property `in<>-suff` and providing ACL2 with a hint to construct the necessary witness. For example, to prove that polynomial ideals are closed under addition we straightforwardly built an intermediate function computing the witness for `(in<> (+ p q) F)` from `(in<>-witness p F)` and `(in<>-witness q F)`.

Definition 3. The congruence induced by an ideal I, denoted as $\equiv_I$, is defined as the following relation: $p \equiv_I q \iff p - q \in I$.

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

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

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

5. Polynomial reductions

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

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

Some of the key results we need to formally verify Buchberger's algorithm (for example Newman's Lemma) can be proved by reasoning about reductions in an abstract setting, and applying them to polynomial reductions as a particular case. By an *abstract reduction* we simply mean a binary relation. In Ruiz-Reina et al. (2002), a library of results about abstract reductions was developed and applied to the particular case of term rewriting systems. In this library, a result about abstract reductions is usually formalized introducing the reduction by means of an `encapsulate`, assuming some properties about it, and proving the desired theorem from them. Functional instantiation can then be used to apply the proved result to a particular reduction that meets the assumed properties.

Thus, we have formalized polynomial reductions in this framework for abstract reductions. This approach will allow us to export, by functional instantiation, well-known properties of abstract reductions to the case of polynomial reductions, avoiding the need to prove them from scratch.

In Ruiz-Reina et al. (2002), instead of defining reductions as binary relations, they are defined as the action of *operators* on elements to obtain reduced elements. We follow the same idea to define polynomial reductions. More precisely, the representation of a reduction relation requires defining three functions:

(1) A unary predicate specifying the domain where the reduction is defined. In our case, the domain consists of polynomials, as characterized by the boolean function `k-polynomialp`.

(2) A binary function, `reduction`, computing the application of an operator to a polynomial. In our case, operators are represented by structures[10] $\langle m, c, f \rangle$ consisting of the monomials m and c, and the polynomial f appearing in Definition 4. Notice that we could have just defined operators consisting of c and f, since m can be obtained from them; but we have found closer to the notion of polynomial reduction to exhibit m in the operator representation. The functions accessing these three components are respectively named `o-monomial`, `o-factor` and `o-polynomial`. This is the definition of `reduction`:

```
(defun reduction (p o)
  (let ((c (o-factor o))
        (f (o-polynomial o)))
    (+ p (* c f))))
```

(3) A binary predicate checking whether the application of a given operator to a given object is *valid*. The application of an operator $\langle m, c, f \rangle$ to p is valid if p is a polynomial containing the monomial m, $f \neq 0$ is a polynomial in F and $c = -m/lm(f)$. Notice that the last requirement implies that $lm(f)$ must divide m. This validity predicate is implemented by the function `validp`:

```
(defun validp (p o F)
  (let ((m (o-monomial o))
        (c (o-factor o))
        (fi (o-polynomial o)))
    (and (polynomialp p) (polynomialp fi)
         (monomialp m) (in-monomial m p)
         (member fi F) (not (equal fi (|0|)))
         (dividep (term (first fi)) (term m))
         (equal c (polynomial (MON::- (MON::/ m (first fi))))))))
```

Here the function `dividep` is the divisibility test between terms, `MON::-` and `MON::/` are respectively negation and division of monomials, `term` is the function that accesses the term of a given monomial, `polynomial` constructs a polynomial from a single monomial and `in-monomial` checks if a monomial is in a polynomial. Note that the leading monomial of `fi` is its first monomial, since polynomials are normalized.

These three functions `k-polynomialp`, `reduction` and `validp` are just what we need to define in ACL2 all the concepts related to polynomial reductions, as we will see. Let us begin defining $\leftrightarrow_F$ (the symmetric closure of $\rightarrow_F$). For that purpose, we need the notion of *proof step*[11] to represent the connection of two polynomials by the reduction relation, in either direction (direct or inverse). Each proof step is a structure consisting of four fields: a boolean field marking the step direction, the operator applied, and the polynomials connected (the functions `direction`, `operator`, `elt1` and `elt2` respectively access these four components of a proof step). A proof step is *valid* if one of its elements is obtained by a valid application of its operator to the other element in the specified direction. The function `valid-proof-stepp` checks the validity of a proof step:

```
(defun valid-proof-stepp (step F)
  (let ((p (elt1 step)) (q (elt2 step))
        (o (operator step)) (dir (direction step)))
    (and (implies dir
                  (and (validp p o F)
                       (equal (reduction p o) q)))
```

[10] We use the `defstructure` construct, by Brock (1997), which provides records in ACL2 in a similar way to Common Lisp's `defstruct`.

[11] Here, we use the word "proof" as a synonym of "formal proof in an algebraic proof system". Please, do not confuse these proofs with the ACL2 proofs obtained by the theorem prover.

```
        (implies (not dir)
                 (and (validp q o F)
                      (equal (reduction q o) p))))))
```

The following function formalizes the relation $\leftrightarrow_F$ in ACL2. Note that the `step` argument explicitly introduces the proof step justifying that $p \leftrightarrow_F q$.

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

Next, we define the relation $\overset{*}{\leftrightarrow}_F$ (the equivalence closure of $\rightarrow_F$). This can be described by means of a sequence of concatenated proof steps, which we call a *polynomial proof* or simply a *proof*. Now the `proof` argument explicitly introduces the proof steps justifying that $p \overset{*}{\leftrightarrow}_F q$.

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

We define the relation $\overset{*}{\rightarrow}_F$ (the transitive closure of $\rightarrow_F$) by a similar function, `->*`, just checking in addition that all proof steps are direct. We say that a proof with this shape is a *direct proof*.

The following theorems establish that the congruence $\equiv_{\langle F\rangle}$ is equal to $\overset{*}{\leftrightarrow}_F$, the equivalence closure. This is crucial to connect the results about reduction relations and polynomial ideals, as Section 7 will show.

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

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

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

6. Noetherianity and normal forms

An essential property that will allow us to compute with polynomial reductions is noetherianity:

Definition 5. A reduction relation $\rightarrow$ on a set A is *noetherian* (or *terminating*) if there is no infinite reduction sequence $a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow \cdots$ for $a_i \in A$.

In our case, we are dealing with noetherian reductions:

Theorem 6. *Let F be a finite set of polynomials. Then $\rightarrow_F$ is noetherian.*

Let us explain how we formalized this theorem in ACL2. As we explained in Section 2, the only primitive well-founded relation in ACL2 is o<, defined on objects satisfying o-p, which represents ordinals up to ϵ_0. New user-defined well-founded relations can be introduced in ACL2, based on the following meta-theorem: a relation R on a set A is well-founded if there exists an ordinal embedding $F : A \rightarrow \epsilon_0$ such that $R(x, y)$ implies $F(x) <_{\epsilon_0} F(y)$, where $<_{\epsilon_0}$ denotes o<. Once a relation is proved to be well-founded in this sense (and the corresponding theorem is stored as a `well-founded-relation` rule), it can be used in the admissibility test for recursive functions.

In Medina-Bulo et al. (2002), a polynomial ordering, <, is defined as a lexicographic extension of the monomial ordering[12] and it is proved to be a well-founded relation in the ACL2 sense by providing an explicit ordinal embedding, `polynomial->ordinal`:

```
(defthm <-well-foundedness
  (and (implies (polynomialp p)
                (o-p (polynomial->ordinal p)))
       (implies (and (polynomial p) (polynomial q)
                     (< p q))
                (o< (polynomial->ordinal p) (polynomial->ordinal q))))
  :rule-classes :well-founded-relation)
```

Any reduction relation contained in a well-founded relation will be noetherian in the sense of Definition 5. Thus, the well-founded polynomial ordering < can be used to state the noetherianity of the polynomial reduction. For this purpose, it suffices to prove that the application of a valid operator to a polynomial produces a smaller polynomial with respect to this well-founded relation:

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

Normal forms are also an important concept in our formalization:

Definition 7. A polynomial p is in *normal form* (or is *irreducible*) w.r.t. $\rightarrow_F$ if there is no q such that $p \rightarrow_F q$. Otherwise, p is said to be reducible. A polynomial q is a normal form of p w.r.t. $\rightarrow_F$, denoted as $q = nf_F(p)$, if $p \stackrel{*}{\rightarrow}_F q$ and q is irreducible w.r.t. $\rightarrow_F$.

As a consequence of noetherianity, we will be able to define a function computing a normal form of a polynomial. First, we define a function `reducible`, implementing a reducibility test: when applied to a polynomial, p, and a list of polynomials, F, it returns a valid operator whenever it exists, `nil` otherwise. The following theorems state the main properties of `reducible`:

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

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

[12] The monomial ordering $<_M$ used is just the lexicographic term ordering, though any other admissible ordering would work.

Now it is easy to define a function `nf` computing a normal form of a given polynomial with respect to the reduction relation induced by a given list of polynomials. This function is quite simple: it iteratively tests reducibility and applies valid operators until an irreducible polynomial is found. Note that termination is guaranteed by the noetherianity of the reduction relation, which follows from the well-foundedness of the polynomial ordering and the ACL2 theorem just above Definition 7.

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

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

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

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

Although `nf` is suitable for *reasoning* about normal form computation, it explicitly deals with operators, which constitute a concept of theoretical nature. From a programmer's point of view, an implementation of Buchberger's algorithm should not explicitly use this concept, as it has been introduced just for *specification* purposes. Moreover, it would be overly inefficient. At this point, we describe the polynomial reduction function red^*_F used in our implementation of Buchberger's algorithm. This function (whose definition we omit) does not make any use of operators but it is modeled from the closure of the set extension of another function, *red*, which takes two polynomials as its input and returns the result of reducing the first polynomial with respect to the second one. The following theorem shows the equivalence between nf_F and red^*_F:

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

With this result, we can translate all the properties proved about `nf` to `red*`. This is typical in our formalization: we use one function for reasoning while another is used for computing, translating properties from the former to the latter by proving equivalence theorems. For example, using this technique we proved the stability of the ideal with respect to `red*`, as established by the following theorem:

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

```
(iff (in<> (red* p F) F)
     (in<> p F))))
```

7. Gröbner bases

The computation of normal forms with respect to a given ideal can be seen as a generalized polynomial division, where remainders play the role of normal forms. The ideal membership problem can be solved by taking this into account: we could compute the normal form and check for the zero polynomial. Unfortunately, it is possible that, for a given basis F, a polynomial in $\langle F\rangle$ cannot be reduced to the zero polynomial. For example, consider the lexicographic term ordering induced by $x > y > z$ and the basis $F = \{x + y, x + z\}$. It is clear that $y - z \in \langle F\rangle$, though $y - z$ cannot be reduced to 0 with respect to F; in fact, it cannot be reduced at all. However, we can connect it to the zero polynomial by using inverse reductions, for example: $y - z \leftarrow_F x + y \rightarrow_F 0$. Thus, $y - z \overset{*}{\leftrightarrow}_F 0$ and therefore $y - z \equiv_{\langle F\rangle} 0$.

Gröbner bases are precisely those finite sets of polynomials whose ideal membership problem can be decided by checking reducibility to the zero polynomial:

Definition 8. A finite set of polynomials G is a Gröbner basis of the ideal generated by F if $\langle G\rangle = \langle F\rangle$ and $(p \in \langle G\rangle \iff p \overset{*}{\rightarrow}_G 0)$.

The purpose of Buchberger's algorithm is to compute a Gröbner basis for the ideal generated by a given finite set of polynomials. But before dealing with the algorithm, we need to prove some key properties.

7.1. Gröbner bases and s-polynomials

The property of being a Gröbner basis can be reformulated in the context of the theory of abstract reductions. Let us review some important definitions:

Definition 9. Let $\rightarrow$ be a reduction relation on a set A. We say that $u, v \in A$ are joinable (denoted as $u \downarrow v$) if there exists y such that $u \overset{*}{\rightarrow} y$ and $v \overset{*}{\rightarrow} y$. A reduction relation has the *Church–Rosser property* if every two objects u and v such that $u \overset{*}{\leftrightarrow} v$ are joinable.

It is clear from the last ACL2 theorems displayed in Section 5 that G is a Gröbner basis of the ideal generated by F if $\langle G\rangle = \langle F\rangle$ and $\rightarrow_G$ has the Church–Rosser property. Furthermore, the Church–Rosser property can be localized when the reduction is noetherian: in that case an equivalent property is *local confluence*, as shown by Newman's Lemma:

Definition 10. Let $\rightarrow$ be a reduction relation on a set A. We say that $\rightarrow$ is *locally confluent* if for all $x, u, v \in A$ such that $u \leftarrow x \rightarrow v$, then $u \downarrow v$.

Theorem 11 (*Newman's Lemma*). *A noetherian and locally confluent reduction relation has the Church–Rosser property.*

Thus, in order to prove that a finite basis G is a Gröbner basis, it is sufficient to check that the corresponding reduction relation is locally confluent, since polynomial reduction relations are noetherian. This still requires, in principle, to check an infinite number of polynomial pairs for joinability. But the key point is that local confluence can be decided by checking only finitely many "critical situations", an idea analogous to the notion of *critical pair* that appears in the area of *term rewriting systems*, see Baader and Nipkow (1998). In fact, there is a close connection between Knuth–Bendix's completion algorithm for term rewriting systems and Buchberger's algorithm. In the latter, this notion of critical local divergence is characterized through the notion of s-polynomial.

Definition 12. Let p and q be polynomials. Let m, m_1 and m_2 be monomials such that $m = \mathrm{lcm}(lm(p), lm(q))$ and $m_1 \cdot lm(p) = m = m_2 \cdot lm(q)$. The *s-polynomial* induced by p and q is defined as $s\text{-}poly(p, q) = m_1 \cdot p - m_2 \cdot q$.

Theorem 13. *Let* $\Phi(F) \equiv \forall p, q \in F$ *s-poly*$(p, q) \overset{*}{\rightarrow}_F 0$. *The reduction induced by F is locally confluent if* $\Phi(F)$ *holds.*

Let us first deal with the formalization and proof of this theorem in ACL2, which was one of the most difficult tasks in our work. First, note that it cannot be expressed, as stated, with a single quantifier-free formula in the ACL2 logic, because of the universal quantifier in its hypothesis, $\Phi(F)$. One possibility to overcome this problem would be to define a function for computing the (finite) set of s-polynomials generated from all the pairs of polynomials in F, and another for checking the reducibility of all the s-polynomials to 0. Note that reduction to the zero polynomial is based on the existence of a proof connecting the polynomial to 0, and therefore it could be expressed by a function defined with an existential quantifier, via `defun-sk`.

Instead of this, we prefer to state the hypothesis of Theorem 13 using the encapsulation principle, making explicit the function witnessing the proof that connects an s-polynomial to 0. One advantage of this approach will be clear in the next subsection, where that witness function is shown to be an essential component of the ACL2 proof obtained. Another advantage is that, in this stage of the formalization, we avoid dealing with the function computing all the s-polynomials, postponing this to the implementation of the algorithm. The `encapsulate` stating the hypothesis of the theorem (we omit the local witnesses and some non-essential technical details) is:

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

With this `encapsulate`, we are assuming that we have a list of polynomials provided by the 0-ary function `F`, with the property that every s-polynomial formed with pairs of elements of `(F)` is reduced to `(|0|)`, where `(F)` and `(|0|)` are the applications of `F` and `|0|`. This reduction is justified by another function, `s-polynomial-proof`, returning the corresponding sequence of proof steps. We insist that `F` and `s-polynomial-proof` are not completely defined: we are *only* assuming the property `|Phi(F)|` about them.

Now, the conclusion of Theorem 13 is established as follows:

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

This theorem needs some explanation. Note that local confluence can be reformulated in terms of the "shape" of the involved proofs: a reduction is locally confluent if, and only if, for every *local peak* proof (i.e., of the form $p \leftarrow r \rightarrow q$) there exists an equivalent *valley* proof (that is, of the form $p \overset{*}{\rightarrow} s \overset{*}{\leftarrow} q$). It is straightforward to define in ACL2 the functions `local-peakp` and `valleyp` checking if a proof has the expected shape. Note again that the valley proof in the above theorem is given by a function, `transform-local-peak-F`, which computes an equivalent valley proof from a given local peak proof. This function must be defined and the theorem `|Phi(F) => local-confluence(->F)|` above proved.

The definition of the function `transform-local-peak-F` is very long and, not surprisingly, follows the same structure as the classical proof of the result. In only one of its cases (the one dealing with "critical overlaps"), `s-polynomial-proof` is used as an auxiliary function, reflecting in this

way where the assumption about $\Phi(F)$ is necessary. We leave a more detailed explanation of this critical overlap case for the next subsection.

As we said before, Newman's Lemma has already been proved in ACL2 (Ruiz-Reina et al., 2002). Thus, we can reuse this general result, instantiating every function used in the abstract formulation with the corresponding function of the polynomial case, and applying the derived rule of inference of functional instantiation. For example, in the proof of Newman's Lemma, an abstract reduction relation is assumed to be locally confluent, formulated in terms of "proof shapes" as above, and proved to have the Church–Rosser property. When we functionally instantiate the result for the particular case of the polynomial reduction associated to `(F)`, a proof obligation is generated, requiring local confluence of the polynomial reduction. But this is precisely what was proved by the theorem `|Phi(F) => local-confluence(->F)|` above.

Thus, we can apply local confluence, Newman's Lemma, and the noetherianity of the polynomial reduction relation proved in Section 6, to show that if $\Phi(F)$, $p \overset{*}{\leftrightarrow}_F q \iff nf_F(p) = nf_F(q)$. Specializing this result with $q = 0$, and using the equality between nf_F and red^*_F, and the equivalence between $\equiv_{\langle F\rangle}$ and $\overset{*}{\leftrightarrow}_F$, it can be easily deduced that if $\Phi(F)$ then F is a Gröbner basis (of $\langle F\rangle$). This is established by the following theorem (notice that `(F)` is still the list of polynomials assumed to have property Φ by the previous `encapsulate`):

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

7.2. The critical overlap case

We have just shown in the preceding subsection how the main property relating s-polynomials and local confluence can be formulated in terms of *proof transformations*. In particular, we needed to define a function `transform-local-peak-F` such that given a local peak proof of the form $p \leftarrow r \rightarrow q$ as input, returns a valley proof joining p and q. In this subsection we describe this function, explaining in more detail how the critical overlap case is managed.

The definition of the function `transform-local-peak-F` follows the same case distinction as in the classical proof, depending on the monomials of p where both reductions take place. This is the main body of the function:

```
(defun transform-local-peak-F (proof)
  (let ((m1 (o-monomial (operator (first proof))))
        (m2 (o-monomial (operator (second proof)))))
    (cond ((equal (term m1) (term m2))
           (transform-local-peak-F-= proof))
          ((TER::< (term m1) (term m2))
           (transform-local-peak-F-< proof))
          (t
           (transform-local-peak-F-> proof)))))
```

Note that the monomials where the reductions take place can be extracted respectively from the operators of the first and second step of the proof. Three cases arise respectively dealing with the three possibilities of relating these two monomials with respect to `TER::<`, the ordering between terms (each of them is managed by the corresponding auxiliary function). We will describe in more detail the auxiliary function `transform-local-peak-F-=`, dealing with the case where both reductions take place in the same monomial of p. For this case, two lemmas are needed in the classical proof of the result:

Lemma 14. *Let p and q be polynomials, and F a finite set of polynomials such that $p - q \overset{*}{\rightarrow}_F 0$. Then $p \downarrow_F q$.*

Lemma 15. *Let $m \neq 0$ be a monomial, p and q polynomials, and F a finite set of polynomials such that $p \xrightarrow{*}_F q$. Then $m \cdot p \xrightarrow{*}_F m \cdot q$.*

Again, these two lemmas can be reformulated in terms of proof transformations since both establish the existence of a target proof given an input proof. In our ACL2 formalization this means that we have to respectively define two functions, `lemma-14-proof` and `lemma-15-proof`, returning these target proofs and prove that the proofs returned meet the intended properties. We defined both functions and proved the following about them:

- If `proof` is a direct proof connecting $p - q$ to 0, then `(lemma-14-proof p q proof)` is a valley proof connecting p and q.
- If `proof` is a direct proof connecting polynomials p and q, and m is a non-null monomial, then `(lemma-15-proof m proof)` is a direct proof connecting $m \cdot p$ and $m \cdot q$.

The functions `lemma-14-proof`, `lemma-15-proof` and `s-polynomial-proof` are just the ingredients we need for the following proof transformer, `transform-local-peak-F-=`, which builds an equivalent valley proof for every critical local peak proof in `(F)`:

```
(defun transform-local-peak-F-= (proof)
  (let ((fi (o-polynomial (operator (first proof))))
        (fj (o-polynomial (operator (second proof))))
        (m (o-monomial (operator (first proof)))))
    (lemma-14-proof (elt1 (first proof))
                    (elt2 (second proof))
                    (lemma-15-proof (coeff-lcm m fj fi)
                                    (s-polynomial-proof fj fi)))))
```

See how the definition of this function reflects the intuitive idea that the s-polynomials of a basis represent the critical divergences of its associated reduction relation and that all the possible divergences can be solved if s-polynomials are reducible to zero. Given a local peak proof, the polynomials `fi` and `fj` of the basis used in the reduction steps are extracted, together with the monomial `m` where both reductions take place. By assumption, `(s-polynomial-proof fj fi)` returns a direct proof connecting *s-poly*(f_j, f_i) to the zero polynomial. With a suitable monomial (given by `(coeff-lcm m fj fi)`) and using the function `lemma-15-proof` we obtain a direct proof connecting $p - q$ to zero. And finally using the function `lemma-14-proof`, we obtain a valley proof connecting p and q.

Finally, we must define the function `transform-local-peak-F-<` and its symmetric, `transform-local-peak-F->`, to deal with the non-overlapping cases. We omit their descriptions, but it is interesting to point out that in none of these cases the function `s-polynomial-proof` is needed. Having dealt with the three possible types of local peaks, the function `transform-local-peak-F` is defined as above, and the theorem `|Phi(F) => local-confluence(->F)|` of the previous subsection proved.

8. Buchberger's algorithm

The ACL2 theorem at the end of Section 7.1 shows that an arbitrary G such that $\Phi(G)$ is a Gröbner basis. The idea is to extend a given finite set of polynomials F_1, obtaining a *finite* sequence of sets $F_1, \ldots, F_n$ such that $\langle F_1 \rangle = \cdots = \langle F_n \rangle$ and $G = F_n$ satisfies Φ. Hence, G is the desired Gröbner basis. This is the purpose of Buchberger's algorithm. We will carry out the above argument in ACL2 as follows. Recall that we introduced `(F)` to be an arbitrary finite set satisfying Φ. Thus, we can functionally instantiate the aforementioned theorem with a substitution binding F to a 0-ary function whose body is the application of Buchberger's algorithm to F_1. In the following, we will develop the formalization of this idea in detail.

Buchberger's algorithm obtains a Gröbner basis of a given finite set of polynomials F by the following procedure: if there is an s-polynomial of F such that its normal form is not zero, then this

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

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

Fig. 1. An ACL2 implementation of Buchberger's algorithm.

normal form can be added to the basis. This makes it reducible to zero (without changing the ideal), but new s-polynomials are introduced that have to be checked. This *completion* process is iterated until all the s-polynomials of the current basis are reducible to zero.

Fig. 1 presents our ACL2 implementation of Buchberger's algorithm. The main function `Buchberger` computes the initial pairs from a basis (with the auxiliary function `initial-pairs`) and starts the real process, as defined by the main recursive function `Buchberger-aux`. The auxiliary function `pairs` returns the ordered pairs built from its first argument and every element in its second argument.

As all ACL2 functions must be total and we need to deal with polynomials with a fixed set of variables to ensure termination of the function, we have to explicitly check that the arguments remain in the correct domain, as checked by functions `k-polynomialsp` and `k-polynomial-pairsp`. We will comment more about these "type conditions" in Section 9.

We now describe our ACL2 formalization and proofs of the main properties of the function `Buchberger`. That is, we show that it terminates and computes a Gröbner basis of the ideal generated by its input basis.

8.1. Termination

Termination of the function `Buchberger` is not trivial at all. A suitable decreasing ordinal measure on the arguments has to be explicitly supplied, in order to be admitted by the principle of definition.

The classical termination proof of Buchberger's algorithm is mainly based on the following result, known as Dickson's Lemma:

Theorem 16 (*Dickson's Lemma*). *Let $k \in \mathbb{N}$ and $\{m_n : n \in \mathbb{N}\}$ an infinite sequence of terms in a set of k variables. Then, there exist indices $i < j$ such that m_i divides m_j.*

Taking Dickson's Lemma into account, we can give the following informal argument justifying the termination of the process carried out by the function `Buchberger`:

(1) In the first recursive branch, the first argument keeps unmodified while the second argument structurally decreases, since one of its elements is removed.
(2) In the second recursive branch, the first argument decreases in a certain well-founded sense, despite the inclusion of a new polynomial. Note that the polynomial added to the basis, h, is not 0 and that it cannot be reduced by F. Consequently, its leading term is not divisible by *any* of the leading terms of the polynomials in F. Considering the sequence of leading terms of the

polynomials successively added to the basis, Dickson's Lemma ensures that this situation cannot happen an infinite number of times.

We have formalized this intuitive argument in the ACL2 logic. First of all, we need a proof of Dickson's Lemma. Two different ACL2 formalizations of this result have been given in Martín-Mateos et al. (2003) and Sustik (2003). In both cases it has been proved by providing an ordinal measure on finite sequences of terms which decreases every time a new term not divisible by any of the previous terms in the sequence is added. Let us make this precise.

The following function `exists-divisible` takes as input a list of terms `LT` and a term `term` and checks if `term` is divisible by at least one of the terms of `LT`:

```
(defun exists-divisible (LT term)
  (if (endp LT)
      nil
    (or (dividep (first LT) term)
        (exists-divisible (rest LT) term))))
```

Essentially, what has been proved in both formalizations is that an ordinal measure function, `Dickson-measure`, can be defined on lists of terms (and on the number k of variables) to hold the following properties:

```
(defthm Dickson-measure-ordinal
  (o-p (Dickson-measure LT k)))

(defthm Dickson-measure-decreases
  (implies (and (natp k)
                (k-termp term) (k-term-listp LT)
                (not (exists-divisible LT term)))
           (o< (Dickson-measure (cons term LT) k)
               (Dickson-measure LT k))))
```

That is, when a new term (in k variables) is added to a sequence of terms (in k variables), this ordinal measure decreases provided that the added term is not divisible by any of the terms in the sequence. The difference between the formalizations given in Martín-Mateos et al. (2003) and Sustik (2003) relies on the different approaches to the definition of `Dickson-measure`, based on totally different ideas. We strongly encourage the reader to read both interesting formalizations in detail, formalizations that have been used interchangeably in our termination proof of Buchberger's algorithm.

Having defined `Dickson-measure`, the definition of a measure to show termination of Buchberger's algorithm is quite straightforward. It suffices to define a lexicographic combination of `Dickson-measure` (applied to the sequence of leading terms of the elements of the first argument) and the length of the second argument.

More precisely: if μ is the ordinal measure of the leading terms of the polynomials in F and λ is the cardinal of C, we define the function `Buchberger-measure` to return the ordinal $\omega^{\mu} + \lambda$, which essentially means that the measures on F and C are compared lexicographically. We omit here the, rather technical, definition of this function.

The final part in our termination proof is to show that in the second recursive call, the leading term of the new polynomial added is not divisible by any of the leading terms of the polynomials of the current basis (as a consequence of its irreducibility). Having proved this, we simply apply the properties of `Dickson-measure` to conclude that `Buchberger-measure` is an ordinal measure on the arguments of `Buchberger-aux` which strictly decreases in any of its two recursive calls. Thus, the definitions of `Buchberger-aux` and `Buchberger` are admitted in the logic, allowing us to reason about its partial correctness.

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

Fig. 2. Polynomial proofs in Buchberger's algorithm.

8.2. Partial correctness

In order to prove that the function `Buchberger` implementing Buchberger's algorithm indeed computes a Gröbner basis G from F, and taking into account the results of the previous section, we just have to guarantee that $\langle F\rangle = \langle G\rangle$ and that G satisfies the Φ property. The following theorems formalize these properties in ACL2:

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

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

The first theorem is an easy consequence of the stability of an ideal with respect to the reduction function `red*`, shown at the end of Section 6.

The statement of the second theorem deserves some comments. Our ACL2 formulation of Theorem 13 defines the property $\Phi(F)$ as the existence of a function such that for every s-polynomial of F, it computes a sequence of direct proof steps justifying its reduction to `(|0|)` (assumption `|Phi(F)|` in the `encapsulate` of the previous section). Thus, if we want to establish the property Φ for a particular basis (the basis returned by `Buchberger`, in this case), we must explicitly define such a function and prove that it returns the desired proofs for every s-polynomial of the basis.

In this case, this function is `|Phi(Buchberger(F))|-proof`. Fig. 2 presents its main auxiliary component, the function `Buchberger-proofs`, which collects, every time a new s-polynomial is examined, the corresponding proof justifying the reduction of the s-polynomial to zero.

Note that, not surprisingly, this function has a recursive scheme very similar to `Buchberger-aux`. In fact it does the same computation process but additionally collects in its extra arguments `pairs`

and `proofs`, respectively, the pairs of polynomials analyzed and the proof justifying that the corresponding s-polynomial reduces to zero.

The auxiliary function `nf-proof` obtains a proof justifying the connection of a polynomial to its normal form. If an s-polynomial reduces to zero, that is the proof collected. Otherwise, a final proof step reducing the s-polynomial to zero by itself is added (by function `h-proof`).

Now `(|Phi(Buchberger(F))|-proof p q F)` is simply defined to take, from the proofs collected by `Buchberger-proofs`, the corresponding proof justifying the reduction to zero of `(s-poly p q)`.

At this point, we would like to emphasize again the constructive character of our formalization. Although `Buchberger-proofs` is *only used for reasoning*, it explicitly constructs a justification of the main property we are claiming about `Buchberger-aux`. And this is done by replaying its computation process.

8.3. Deciding ideal membership

Finally, we can compile all the previous results to obtain the following main theorem:

Theorem 17. *Let F be a finite set of polynomials. Then our* ACL2 *implementation of Buchberger's algorithm returns a Gröbner basis of ⟨F⟩ when F is given as its input.*

As the admission of the function `Buchberger` in the ACL2 logic implies its termination, the ACL2 formalization of this theorem reduces to:

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

where `imdp` is the function defined to simply check whether a given polynomial reduces to the zero polynomial with respect to the Gröbner basis returned by the implementation of Buchberger's algorithm:

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

Note that the above theorem can be seen as stating the soundness and completeness of a decision procedure for the ideal membership problem, given by the function `imdp`. It is an easy consequence of the correctness of `Buchberger` and the theorem `|Phi(F) => (p in <F> <=> red*(p, F) = 0)|` in Section 7. We have just to functionally instantiate this theorem replacing the 0-ary function `F` by the function `(lambda () (Buchberger F))`, which is the 0-ary λ-abstraction of `Buchberger`, and `s-polynomial-proof` by `|Phi(Buchberger(F))|-proof`.

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

9. Conclusions

We have shown how it is possible to use the ACL2 system in the formal development of Computer Algebra algorithms by presenting *verified executable* Common Lisp *implementations* of both a simple version of Buchberger's algorithm and a decision procedure for the ideal membership problem based on it.

It is interesting to point out that all the theory needed to prove the correctness of the algorithm has been developed in the ACL2 logic, in spite of its apparently limited expressiveness. Results about *coefficients fields*, *polynomial rings*, *polynomial ideals*, *abstract reductions*, *polynomial reductions*, *ordinal measures* and *Gröbner bases* had to be formalized before reasoning about the properties of the

algorithm. Amazingly all these notions and their related properties fit quite well in the first-order ACL2 logic, which has not been specially designed for the formalization of mathematical theories.

We have benefited from work previously done in the system. In particular, all the results about abstract reductions were originally developed for a formalization of rewriting systems (Ruiz-Reina et al., 2002). Also, two independent proofs of Dickson's Lemma were available in order to prove termination of the main algorithm. We believe that this is a good example of how different formalizations can be reused in other projects, provided that the system offers a minimal support for it. *Encapsulation* plus *functional instantiation* in ACL2 provide a good abstraction mechanism that allows to reuse previous results in a modular way. At the same time this mechanism can be understood as a means of overcoming some limitations inherent to the first-order character of the logic allowing a certain kind of second-order reasoning (at least, in appearance).

The notion of *polynomial proof* is key to our formalization. This notion is implicit in the hand proof described in Baader and Nipkow (1998), but it is made explicit in our formalization, since formulas in the ACL2 logic do not have existential quantifiers. We defined a number of functions that can be seen as *proof transformers*: they take proofs as inputs and transform them to obtain a witness proof with the desired properties. But it has to be clear that these polynomial proofs represent an auxiliary concept, only needed for reasoning about the properties of the algorithm. The target verified implementation does not deal with proofs and it is implemented as any programmer would do, without bothering about automated reasoning concerns.

However, we have introduced *existential quantifiers* in our formalization of polynomial ideals to reflect common mathematical practice when modeling ideal membership. In fact, quantifiers are introduced through *Skolem functions*, another place where encapsulation plays its role in ACL2. From an automated reasoning point of view existential quantifiers pose a problem: they allege the existence of objects satisfying certain properties but offer no clue about how to build them. As ACL2 is aimed to the automation of reasoning, it offers a poor support for quantifiers, at least when compared to other systems with more expressive logics.

9.1. Quantitative data about the proof

As for the proof effort, Tables 1–5 show the number of lines, definitions, theorems and hints needed during the different stages of the verification process. These numbers may give a rough idea of the complexity of the formalized theories and the degree of automation of the proofs obtained. In most cases, each theorem or lemma is automatically proved performing an induction step and simplification, or just the latter. Simplification is generally accomplished by rewriting using previously proved lemmas and by application of the built-in decision procedures.

As for the user interaction required, we provided 169 definitions and 560 lemmas to develop a theory of polynomials (although this includes more than those strictly needed here) and 109 definitions and 346 lemmas for a theory of Gröbner bases and Buchberger's algorithm. All these lemmas were proved almost automatically. It is worth pointing out that of the 333 lemmas proved by induction, only 24 required a user-supplied induction scheme, which are less than 8% of the total inductions. Other lemmas needed a hint about the convenience of using a given instance of another lemma or keeping a function definition unexpanded. Only 9 functions required a hint for their termination proofs.

Finally, more than 80 hints were devoted to functional instantiations. This is positive as each functional instantiation represents the reuse of more abstract results previously established. These represent almost 20% of the total hints and affect roughly 9% of the theorems. This means that 9% of the theorems are obtained "for free", the only work of the user being to identify them as functional instances of other theorems.

We would like to remark that although polynomial properties seem trivial to prove, this is not the case (Medina-Bulo et al., 2000, 2002). It seems that this is not because of the simplicity of the ACL2 logic: this is the price to pay for being formal. In contrast, in well-known Algebra treatises, these properties are usually reduced to the univariate case or their proofs are partially sketched and justified "by analogy". In some cases, the proofs are even left as an exercise.

Table 1
Quantitative data about the formalization of abstract polynomials (i).

Book	Lines	Definitions	Theorems	Hints
Coefficients. An abstract abelian coefficient ring represented by an encapsulation. The set of ACL2 numbers with its standard interpretation serves as a model of the generated logical theory.	185	7	36	12
Terms. An abstract commutative term monoid equipped with a well-founded order and represented by an encapsulation. Proper lists of ACL2 natural numbers with element-wise addition and lexicographical ordering serve as a model of the generated logical theory. Well-foundedness is established by an ϵ_0-ordinal embedding.	115	7	16	2
Monomials. Coefficient-term pairs. A monomial equivalence is defined such that two monomials with a zero coefficient are considered equivalent.	135	14	24	8
Polynomials. Abstract representation of polynomials by proper lists of monomials containing abstract coefficients and terms.	17	5	1	0
Polynomial normal form. Normalization function. Induced polynomial equivalence as equality of normal forms. Related important properties like normalization idempotence.	145	8	23	4
Polynomial addition. Polynomial addition as concatenation of the underlying monomial lists. Polynomials with polynomial addition form a commutative monoid.	46	1	11	3
Polynomial addition congruences. Polynomial equality is congruent with polynomial addition.	22	0	5	2
Polynomial negation. Polynomial additive inverse. Polynomials with polynomial addition and negation form a commutative group.	81	1	13	8
Polynomial multiplication. External and internal product. Polynomials with polynomial multiplication form a commutative monoid. Distributivity of polynomial multiplication over polynomial addition.	105	5	19	9
Polynomial multiplication congruences. Polynomial equality is congruent with polynomial external and internal multiplication.	120	1	18	4
Total	971	49	166	52

Table 2
Quantitative data about the formalization of abstract polynomials (ii).

Book	Lines	Definitions	Theorems	Hints
Normalized polynomials. Normalized polynomials defined on top of unnormalized polynomials by using the normalization function. Lifting ring properties from the unnormalized representation to the normalized representation.	167	9	26	16
Polynomial order. Extending the monomial ordering to polynomials. Well-foundedness is established by embedding polynomials into ϵ_0-ordinals.	35	2	6	0
Total	202	11	32	16

9.2. Related work and comparison

Work aimed to obtaining verified implementations of Buchberger's algorithm can be classified according to the approach followed. In the *integrated approach* a constructive proof of the existence of Gröbner bases is developed in a constructive logic and the algorithm is extracted from the proof

Table 3
Quantitative data about the formalization of rational polynomials (i).

Book	Lines	Definitions	Theorems	Hints
Rationals. Instantiation of abstract coefficients with ACL2 rational numbers. Division operation and its properties.	281	9	61	27
Terms. Instantiation of abstract terms with proper lists of ACL2 natural numbers.	137	9	20	3
Term division. Divisibility, division and least common multiple on terms.	125	3	27	0
Monomials. Instantiation of abstract monomials to get rational-term pairs using concrete terms.	176	16	38	7
Polynomials. Instantiation of abstract polynomials to get polynomials with rational coefficients represented by proper lists whose components are concrete monomials.	46	5	1	1
Polynomial normal form. Instantiation of the normalization function to work with rational polynomials.	147	7	10	7
Polynomial addition. Instantiation of polynomial addition to work with rational polynomials.	63	2	5	5
Polynomial addition congruences. Polynomial equality is congruent with rational polynomial addition. Functional instantiation of abstract polynomial counterparts.	26	0	10	10
Polynomial negation. Instantiation of polynomial negation to work with rational polynomials.	52	2	3	2
Polynomial multiplication. Instantiation of polynomial multiplication (external and internal product) to work with rational polynomials.	82	5	9	7
Polynomial multiplication congruences. Polynomial equality is congruent with rational polynomial external and internal multiplication. Functional instantiation of abstract polynomial counterparts.	11	0	4	4
Normalized polynomials. Instantiation of abstract normalized polynomials to get normalized polynomials with rational coefficients.	267	12	44	22
Polynomial order. Instantiation of the polynomial order to work with rational polynomials.	95	3	5	5
Total	1508	73	237	100

itself in a more or less automatic way. In the *external approach* the algorithm is first constructed and then a computational (though, possibly classical) logic is used to prove its termination and partial correctness.

We have followed an external approach. Each function is implemented in ACL2 and it is admitted once its termination has been proved. Following each function definition, theorems establishing their intended properties are stated and proved.

Coquand and Persson (1999) present an integrated development of Buchberger's algorithm in Martin-Löf's type theory. Although this line seems promising, the formalization is not complete. However some formal proofs have been obtained in AGDA.

Rudnicki et al. (2001) propose to formalize a significant fragment of commutative Algebra in MIZAR. Although MIZAR is not the kind of system where algorithms can be explicitly written, it is specially designed to ease the development of mathematical theories in a modular way. In their work, the authors plan to build a *verification condition generator* (VCG) to obtain proof obligations in the MIZAR language from code implementing Buchberger's algorithm in a programming language.

Table 4
Quantitative data about the formalization of rational polynomials (ii).

Book	Lines	Definitions	Theorems	Hints
Parametric definitions. Macros `defun<k>` and `defun-sk<k>` to define functions with an implicit parameter *k*.	81	15	0	0
Membership. Functions checking membership of monomials to polynomials, and of polynomials to polynomial lists. Properties.	312	3	57	25
Uniform polynomials. Uniformity, polynomials with *k* variables and their properties. Previously defined polynomial operations preserve uniformity.	410	18	68	47
Total	803	36	125	72

Later, these proof obligations could be discharged once the necessary theories have been developed. In Schwarzweller (2006) a rich theory of Gröbner bases is presented, but the external construction of Buchberger's algorithm and a matching VCG suitable for Mizar integration is still pending.

As we mentioned in the introduction, a relevant complete formalization of Buchberger's algorithm is L. Théry's machine-checked implementation in Coq (Théry, 2001), which follows an external approach too. A comparison with this work follows.

An obvious difference between our formalization and Théry's comes from the fact that we are using quite a different system. This has implications on the user interaction and on the logic used. First, the way we lead the prover to the desired mechanical proof is completely different. Coq is a proof checker: the proofs are mainly introduced by the user and machine-checked by the system. In contrast, ACL2 contains a mechanical theorem prover: it essentially applies the same proof techniques to every conjecture submitted by the user, automatically discovering a proof when it succeeds. But, as we remarked in Section 2, the role of the user is important: she has to incrementally build a suitable collection of lemmas in order to make each proof attempt succeed.

Second, the logic of ACL2 is less expressive than the logic of Coq, which is a higher-order logic based on type theory. Nevertheless, the expressiveness of the logic has not been a serious drawback in our formalization, since all the mathematical concepts in the theories developed can be formulated quite naturally in the ACL2 logic. To this respect, the main differences with Théry's work come from the fact that different source hand proofs were formalized (Baader and Nipkow (1998) in our case and Geddes et al. (1992) for the Coq proof).

Théry's formalization does not deal with abstract reductions, and it is specifically focused on polynomial reductions. We reused some previously proved results about abstract reductions (the most important being Newman's Lemma) and used functional instantiation to translate these results into the polynomial case. In this sense, we think that our proof is more general and reusable. Another difference is related to Dickson's Lemma: unlike in our formal proof, the proof of Dickson's Lemma in Théry's formalization is non-constructive and needs a Coq axiom.[13]

We think that the main advantage of our approach is that we do not formalize just an algorithm but an implementation. This implementation is compliant with Common Lisp, a real programming language, and can be directly executed in ACL2 or in any compliant Common Lisp. Taking into account that Lisp is the language of choice for the implementation of many CAS, like Macsyma and Axiom, this is not just a matter of theoretical importance but also a practical one. And it is also remarkable that in ACL2 we can reason directly about the same program that will be eventually used for execution. In contrast, in Théry (2001) an executable implementation in the OCaml language is extracted from the verified algorithm defined in Coq, extraction which requires some manual intervention.

An approach related to Théry's work is presented by Jorge et al. (2009), where the certification of some properties of an efficient Caml program for Gröbner basis computation is considered. Two

[13] Nevertheless, H. Persson obtained a constructive proof of Dickson's Lemma (Persson, 2001), integrating it with Théry's work to obtain a fully constructive proof of Buchberger's algorithm.

Table 5
Quantitative data about the formalization of Buchberger's algorithm.

Book	Lines	Definitions	Theorems	Hints
Polynomial reductions. Reduction relations on polynomials and their main properties.	123	17	7	3
Noetherianity. Noetherianity of the reduction relation on polynomials.	278	2	35	25
Normal form. Normal form with respect to reductions developed with operators.	88	4	9	4
Normal form computation. Reduction functions used in the implementation of Buchberger's algorithm and their equivalence with normal forms.	188	8	37	18
S-polynomials. Definition of s-polynomial and the proofs of its fundamental properties.	54	2	7	5
Buchberger's algorithm. ACL2 implementation of Buchberger's algorithm. Termination proof.	183	12	19	8
Ideals. Polynomial ideals, ideal membership and fundamental properties.	152	8	26	8
Ideal congruence. Congruence induced by an ideal and its agreement with the equivalence relation.	602	18	51	31
Reduction stability. Ideal stability under the reduction set extension and its closure.	64	0	10	2
S-polynomial stability. Ideal stability under the computation of s-polynomials.	18	0	1	1
Suffix stability. Suffix of a sequence of polynomials and its basic properties. Ideal membership is preserved under a suffix extension of the basis.	67	2	14	2
Buchberger's algorithm stability. Ideal stability under Buchberger's algorithm. Several properties stating that the ideal remains invariant under the operations which compose the algorithm are used.	95	3	20	2
Confluence. What is proved here is that whenever the s-polynomials of a basis reduce to zero, the induced reduction relation is locally confluent.	687	15	57	25
Gröbner bases. It is proved here that if all the s-polynomials of a basis reduce to zero, then it is a Gröbner basis. Then, it is proved that all the s-polynomials of the basis computed by Buchberger's algorithm reduce to zero. Finally, it follows that the algorithm computes a Gröbner basis.	560	17	51	31
Decision procedure. Verified decision procedure for the ideal membership problem.	26	1	2	2
Total	3185	109	346	167

approaches are combined to verify these properties: manual proofs that reason directly over the source code of the algorithms and formal proofs about an abstract Coq model for some parts of the Caml implementation.

Finally, a new approach is being undertaken by B. Buchberger in the framework of the Theorema project (Buchberger et al., 2006). The idea consists of using *lazy thinking* (Buchberger and Crăciun, 2004) to automate specification-driven *algorithm synthesis*, i.e., the design of provably correct algorithms from their specifications. Lazy thinking had proved successful in deriving simple algorithms, like some sorting algorithms, but the author has shown how powerful the method is

by presenting a roadmap for the complete automated synthesis of his Gröbner bases algorithm. Moreover, Buchberger has shown that this roadmap is feasible so the synthesis is possible in THEOREMA (Buchberger, 2004).

The process can be seen as a particular case of *theory exploration* or *mathematical knowledge management* (Buchberger et al., 2006). The support of the entire process of mathematical theory exploration is a distinctive goal of THEOREMA. Exploration implies the invention of mathematical concepts, properties about these concepts, problems formulated in terms of them and algorithms to solve these. Purported properties can be proved or disproved and algorithms verified as the exploration proceeds. THEOREMA emphasizes the use of higher-order equational logic as a programming language. This allows for the execution of verified algorithms within the same system where they were verified.

9.3. Final remarks and future work

In Théry (2001), some standard optimizations for the final version of the algorithm are verified. We do not include such optimizations. In fact, efficiency was not our main concern in a first stage. Regarding future work, we are interested in studying how our verified implementation could be improved to incorporate some of the refinements built into the very specialized and optimized (and not formally verified) versions used in industrial strength applications. ACL2 seems specially suitable for this task, since ACL2 programs can be efficiently executed in the underlying "raw" LISP implementation at C-like performance. See the work of Greve, Wilding and Hardin in Kaufmann et al. (2000a), for example.

An obvious improvement in the verified implementation is to avoid the "type conditions" in the body of `Buchberger-aux`, since these conditions are unnecessarily evaluated in every recursive call. But these conditions are needed to ensure termination. Until ACL2 version 2.7, there was no way to avoid this; but since ACL2 version 2.8 that is no longer true, since it is possible for a function to have two different bodies, one used for execution and another for its logical definition: this is done by previously proving that both bodies behave in the same way on the intended domain of the function. The interested reader is referred to Greve et al. (2008), where the `mbe` and `defexec` features of ACL2 are explained in detail. We plan to apply these new features to our definition of Buchberger's algorithm.

Acknowledgements

The authors would like to thank Matyas Sustik and Francisco J. Martín Mateos for their formal proofs of Dickson's Lemma in ACL2. J Strother Moore provided a very productive stay at Austin for the three of us during which some important ideas were developed. We are also grateful to the anonymous reviewers for their careful corrections and useful feedback.

References

Baader, F., Nipkow, T., 1998. Term Rewriting and All That. Cambridge University Press, New York, NY, USA.

Boyer, R.S., Moore, J S., 1998. A Computational Logic handbook, 2nd edition. Academic Press Professional, Inc., San Diego, CA, USA.

Brock, B., December 1997. Defstructure for ACL2 version 2.0. Tech. rep., Computational Logic, Inc.

Buchberger, B., 1970. Ein algorithmisches Kriterium für die Lösbarkeit eines algebraischen Gleichungssystems. Æquationes Math. 4, 374–383.

Buchberger, B., 2004. Towards the automated synthesis of a Gröbner bases algorithm. RACSAM (Reviews of the Spanish Royal Academy of Sciences) 98 (1), 65–75.

Buchberger, B., 2006. Bruno Buchberger's PhD thesis 1965: An algorithm for finding the basis elements of the residue class ring of a zero dimensional polynomial ideal. J. Symbolic. Comput. 41 (3–4), 475–511 (translation by Michael P. Abramson).

Buchberger, B., Crăciun, A., 2004. Algorithm synthesis by lazy thinking: Using problem schemes. In: Petcu, D., Negru, V., Zaharie, D., Jebelean, T. (Eds.), Proc. of SYNASC'04, 6th Int. Symposium on Symbolic and Numeric Algorithms for Scientific Computing. Mirton, Timişoara, Romania, pp. 90–106.

Buchberger, B., Crăciun, A., Jebelean, T., Kovács, L., Kutsia, T., Nakagawa, K., Piroi, F., Popov, N., Robu, J., Rosenkranz, M., Windsteiger, W., 2006. Theorema: Towards computer-aided mathematical theory exploration. J. Appl. Logic 4 (4), 470–504.

Buchberger, B., Winkler, F. (Eds.), 1998. Gröbner Bases and Applications. In: London Mathematical Society Lecture Notes Series, vol. 251. Cambridge University Press, Cambridge, United Kingdom.

Coquand, T., Persson, H., 1999. Gröbner bases in type theory. Lect. Notes Comput. Sci. 1657, 33–46.
Geddes, K.O., Czapor, S.R., Labahn, G., 1992. Algorithms for Computer Algebra. Kluwer Academic Publishers, Norwell, MA, USA.
Greve, D.A., Kaufmann, M., Manolios, P., Moore, J S., Ray, S., Ruiz-Reina, J.L., Sumners, R., Vroon, D., Wilding, M., 2008. Efficient execution in an automated reasoning environment. J. Funct. Program. 18 (1), 15–46.
Jorge, J.S., Gulías, V.M., Freire, J.L., 2009. Certifying properties of an efficient functional program for computing Gröbner bases. J. Symbolic. Comput. 44 (5), 571–582.
Kaufmann, M., Moore, J S., 2001. Structured theory development for a mechanized logic. J. Autom. Reasoning 26 (2), 161–203.
Kaufmann, M., Moore, J S., 2009. ACL2 home page. The University of Texas in Austin.
URL: http://www.cs.utexas.edu/users/moore/acl2.
Kaufmann, M., Moore, J S., Manolios, P. (Eds.), 2000a. Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Publishers, Norwell, MA, USA, pp. 113–136 (Chapter 8).
Kaufmann, M., Moore, J S., Manolios, P., 2000b. Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, Norwell, MA, USA.
Martín-Mateos, F.J., Alonso, J.A., Hidalgo, M.J., Ruiz-Reina, J.L., 2003. A formal proof of Dickson's lemma in ACL2. Lect. Notes Comput. Sci. 2850, 49–58.
Medina-Bulo, I., Alonso-Jiménez, J.A., Palomo-Lozano, F., November 2000. Automatic verification of polynomial rings: Fundamental properties in ACL2. Tech. Rep. TR-00-29, The University of Texas at Austin, Department of Computer Sciences, In: ACL2 Workshop 2000 Proceedings, Part A.
Medina-Bulo, I., Palomo-Lozano, F., Alonso-Jiménez, J.A., April 2002. Implementation in ACL2 of well-founded polynomial orderings. In: Third International Workshop on the ACL2 Theorem Prover and its Applications, ACL2-2002.
Persson, H., September 2001. An integrated development of Buchberger's algorithm in Coq. Tech. Rep. RR-4271, INRIA.
Rudnicki, P., Schwarzweller, C., Trybulec, A., 2001. Commutative algebra in the Mizar system. J. Symbolic. Comput. 32 (1–2), 143–169.
Ruiz-Reina, J.L., Alonso, J.A., Hidalgo, M.J., Martín-Mateos, F.J., 2002. Formal proofs about rewriting using ACL2. Ann. Math. Artif. Intell. 36 (3), 239–262.
Schwarzweller, C., 2006. Gröbner bases — theory refinement in the Mizar system. In: Lecture Notes in Artificial Intelligence, vol. 3863. pp. 299–314.
Steele Jr., G.L., 1990. Common Lisp: The language, 2nd edition. Digital Press, Newton, MA, USA.
Sustik, M., July 2003. Proof of Dickson's lemma using the ACL2 theorem prover via an explicit ordinal mapping. In: Fourth International Workshop on the ACL2 Theorem Prover and its Applications, ACL2-2003.
Théry, L., 2001. A machine-checked implementation of Buchberger's algorithm. J. Automat. Reason. 26 (2), 107–137.