HAL
archives-ouvertes.fr

An Integrated Development of Buchberger's Algorithm in Coq

Henrik Persson

▶ **To cite this version:**

Henrik Persson. An Integrated Development of Buchberger's Algorithm in Coq. RR-4271, INRIA. 2001. inria-00072316

HAL Id: inria-00072316
https://hal.inria.fr/inria-00072316

Submitted on 23 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

An Integrated Development of Buchberger's Algorithm in Coq

Henrik Persson

N° 4271

September 2001

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4271--FR+ENG

An Integrated Development of Buchberger's Algorithm in Coq

Henrik Persson

Thème 2 — Génie logiciel
et calcul symbolique
Projet LEMME

Rapport de recherche n° 4271 — September 2001 — 17 pages

Abstract: We present an integrated formal development of Buchberger's algorithm in Coq, that is we prove constructively the existence of Gröbner bases without explicitly writing the algorithm. This formalisation is based on an external formalisation in Coq by Théry, and an integrated abstract development in Agda. We end by discussing some experiences and differences between the two proof-styles and theorem-provers.

This report was completed in March 2000.

Key-words: Buchberger's algorithm, theorem proving, computer algebra, integrated programming logic, Coq, Agda

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Un développement intégré de l'algorithme de Buchberger en Coq

Résumé : Nous présentons un développement formel intégré de l'algorithme de Buchberger en Coq, c'est-à-dire que nous prouvons constructivement l'existence de bases de Gröbner sans écrire explicitement l'algorithme. Cette formalisation est fondée sur la formalisation externe par Théry et un développement abstrait intégré en Agda. Nous concluons en discutant les expériences et les différences entre les deux styles de développement et les deux systèmes de preuve sur ordinateur.

La rédaction de ce rapport s'est terminée en Mars 2000.

Mots-clés : Algorithme de Buchberger, Preuve de théorèmes, calcul formel, logique intégrée pour la programmation, Coq, Agda

1 Introduction

Girard [Gir86] introduced the two terms *integrated* and *external* programming logic to distinguish between the two ways through which a program could be assured logically correct. In the integrated approach, one writes a constructive proof of the existence of the data that the algorithm computes together with the desired properties, and the program is then seen as a part of the proof. In the external approach, one first writes the program, and then proves it correct. In this approach, the proof may be in a different language, external to the program, and the proof may use non-constructive principles. The proof can be seen as a comment to the program which can be more or less related to the language in which the program is written. For a comparison between the integrated and external approach to program development, see [Dyb90].

Although, Girard states that 'in fact only [the integrated approach] has a theoretical interest', it has been surprisingly little studied. One reason for this is that the main inspiration comes from the Curry-Howard isomorphism [How80] between proofs and programs. In this isomorphism, the proofs are usually restricted to be constructive proofs, but in computer science and mathematics classical proofs are widely used. However, proof theory has come a long way in understanding a wide class of classical proofs important for computer science, see e.g. [WB81]. These results and others have now succesfully been phrased in a more computer science and type theory oriented setting by Coquand [Coq96].

Another reason for the preference of the external approach is efficiency and clarity; one can write the efficient program immediately, and it is simple to reason about properties of the program such as complexity. In the integrated approach, there is however the (so far unrealized) expectation that clever optimizing compilers should be able to use all the formal proof-information which are naturally available. In this setting, the difference between the two approach with regards to optimisation can be seen as the difference between 'proof-optimisation' and 'program-optimisation-using-proofs'.

There has been some positive results by Goad [Goa80a, Goa80b] where he discovered that optimisations on proofs makes certain program optimisations possible which would be impossible by only doing equational optimisation of a program. Without using the proof, the obtained program will be in the same complexity class, whereas by using proof information one can break the complexity barriers. Similar optimisations are of course possible with any programming logic, but there is the belief that the integrated approach gives a more uniform treatment of optimisation. A promising new development towards this goal is the functional programming language Cayenne [Aug98] which extends Haskell [PH96] by adding dependent types. So far, current Cayenne-compilers do not have specific optimisations for dependent types, but we expect this new area of programming language technology to grow in the future (a positive indication of this is the annual 'Dependent Types in Programming' conferences).

So far, there exists several implementations of type theory which directly support the integration of programming logic and functional programming. At INRIA, there is the development of Coq [HKPM97] which allows extraction of formal proofs into CAML [Ler99] and

Haskell from type theoretic proofs, whereas in Sweden, there is Agda [Coq98], which is a type theoretical language almost identical to Cayenne with an interactive proof-environment [Hal98].

At the same time, there has been an increasing effort in developing the next generation of systems for computer algebra. Computer algebra has been a great success-story, and is today commonly used by a wide range of people. But a problem with current computer algebra systems is the lack of support for reasoning, and the lack of declarative specifications of the algorithms [HT98]. For example, they cannot handle side-conditions such as requiring functions to only take arguments with a certain property such as being non-zero. Type theory has been put forward as a promising candidate to be used in such systems to handle this (manifested for example in the joint workshop between the two EU-projects Types and Calculemus [typ98]).

An important aspect when using type theory as a programming logic is the reusability and modification of old proof-developments. This seems especially important for the integrated approach, but is also of general interest since programs and proofs as well as their languages are likely to change during time, and because of the high cost associated to formalisations.

In this paper we describe an integrated formalisation in Coq of Buchberger's algorithm, an essential algorithm in computer algebra with a non-trivial mathematical correctness proof which traditionally is proved classically. The formalisation reuses large parts of an abstract integrated formalisation in Agda [CP99], and an external formalisation in Coq by Théry [Thé98]. We end by discussing the different approaches, the extracted programs, and the difference between these two systems.

2 Buchberger's algorithm

The concept of Gröbner bases together with an algorithm for computing them, was introduced by Buchberger [Buc65, Buc85]. It can be seen as a generalisation of the Euclidian algorithm for computing the greatest common divisor (gcd) of polynomials in several variables. In the case of polynomials in one variable, one can easily decide whether a polynomial f is in the ideal generated by the set of polynomials $F = \{f_1, \ldots, f_n\}$: just compute the gcd of F, since it generates the ideal of F, it is enough to check whether it divides f. This gives many algorithmic solutions to problems concerning polynomials in one variable.

However, in the case of polynomials in several variables, this technique does not work. One problem is that F may not have a single generator of its ideal; hence one needs to define when a polynomial is divided by a set of polynomials. Another more important problem is that this technique is not complete: if F divides f then f is in the ideal of F, but f might be in the ideal of F even though F does not divide f. This is where Gröbner bases come into play; a Gröbner basis G of F is a finite set of polynomials which generates the same ideal and divides f if and only if f is in the ideal of F. There are several thorough presentations of Gröbner bases and their applications [Buc85, Buc98, BW93, CLO97, Frö97].

In [Thé98], Théry presents a complete formalisation of an optimised version of Buchberger's algorithm in Coq. The formalisation is external and self-contained and defines all essential structures, such as monomials and polynomials with the required functions.

From this formalisation he extracted an efficient program computing Gröbner bases [PT]. Since the formalisation is external, the extracted program is almost identical to the desired program.

In [CP99], we presented an *abstract* integrated formalisation; the existence of Gröbner bases was proved *assuming* the existence of structures like monomials and polynomials. An advantage with the abstract approach is that the formalisation is not dependent of the representation of structures such as polynomials and the implementation of operators on these. However, it is incomplete; to execute the certified program, one needs to instantiate all structures, and thus provide the representation and proofs of structures like the polynomial ring. The formalisation [Thé98] was also abstract to a certain extent, but contained furthermore complete instantiations and proofs of the assumptions. In both of these formalisations, the assumptions was *ad hoc*, in the sense that we assumed only what was required; e.g. instead of assuming all the polynomial ring axioms and use these to prove certain lemmas, we assumed these lemmas, and later proved that these lemmas was true for a representation of polynomials without proving all the ring axioms. For reusability and larger formalisations, an approach less ad hoc would be desirable.

We now present a mixture of these two formalisations; we have used the abstract structure of the integrated formalisation [CP99], translated it to Coq, and adopted it to the concrete representations in [Thé98]. The result is a complete integrated formalisation of Buchberger's algorithm in Coq, from which a program could be extracted. Since the formalisation is integrated, it was in fact non-trivial to extract the desired program, and the proof needed to be modified several times.

In this presentation we will take a very abstract view of the algorithm, and show the existence of Gröbner bases under a number of assumptions, we even assume an abstract notion of Gröbner base. We will hence omit important definitions and proofs. These assumptions can be seen defined and proved in the literature mentioned above, and their formal proof can be found in Théry's formalisation. The only proofs we will give in detail are the ones specific to the integrated approach, such as Dickson's lemma and the constructive existence proof of Gröbner bases.

In the rest of this section, we interleave the informal assumptions written in a mathematical style, with the formal assumptions written in typewriter font. Unless stated otherwise, a mathematical constant A corresponds to a formal constant `A`. The formal presentation is complete in the sense that all assumptions are given, with the statements of the main theorems which can be proved under these assumptions.

2.1 Assumptions

In this subsection we enumerate all the assumptions under which the existence of Gröbner bases is proved.

First, we assume a representation of the set of polynomials $K[X_1, \ldots, X_k]$ over a field K. The set of monomials are represented as k-tuples, and we assume a function $mdeg(f)$, the multi-degree of the polynomial f, returning the monomial corresponding to the head monomial of f.

```
Variable poly : Set.
Variable zerop : poly -> Prop.
Hypothesis dec_zerop : (f:poly) {zerop f} + {~ (zerop f)}.

Variable k : nat.
Local trm := (mon k).
Local tdiv:trm->trm->Prop := (mdiv k).
Variable mdeg:poly->trm.
```

Note that the set only comes with a decidable zero-predicate `zerop`, and from the assumptions of a number of variables `k`, we directly define the monomials `trm` and the division `tdiv`. On this abstract level, we do not need to more properties of the polynomial ring or the free monoid of monomials.

We also need a notion of *ideals*, where the ideal of $\{f_1, \ldots, f_n\}$ is written as $\mathrm{Idl}(f_1, \ldots, f_n)$. In the formalisation we assume an equivalence relation `eqI` between lists of polynomials, which captures when two lists generates the same ideal, from this it is direct to define membership of ideals (`Cb`).

```
Variable Cb:poly -> (list poly) ->Prop.
Variable eqI:(list poly)->(list poly)->Prop.
Hypothesis re_eqI : (F:(list poly))(eqI F F).
Hypothesis sy_eqI : (F,G:(list poly))(eqI F G)->(eqI G F).
Hypothesis tr_eqI : (F,G,H:(list poly))(eqI F G)->(eqI G H)->(eqI F H).
Hypothesis cons_eqI : (F,G:(list poly))(f:poly)(eqI G F)->
                          (eqI (cons f G) (cons f F)).
Hypothesis Cb_eqI : (F:(list poly))(f:poly)(Cb f F)->(eqI (cons f F) F).
Hypothesis comm_eqI : (F:(list poly))(f,g:poly)
                          (eqI (cons f (cons g F)) (cons g (cons f F))).
Hypothesis addL_cons_eqI : (F:(list poly))(f:poly)
                              (eqI (cons f F) (addL poly f F)).
```

Here `addL a as` is a function on lists which adds `a` to the end of the list `as`.

For the computation of polynomials, we need a way to reduce a polynomial with regards to a finite set of other polynomials. This function can be defined in several ways, and is fundamentally connected to the notion of Gröbner bases. Essentially, Gröbner bases are exactly those lists of polynomials for which the exact choice of the reduction order in the normalisation function does not matter, and for which this function can be used to decide membership of its ideals.

The normalisation function intuitively corresponds to a generalised version of euclidian division of one polynomial with respect to another; in the case of multivariable polynomials, this is generalised to repeatedly substracting multiples of polynomials. We let $nf(f, F)$ denote an algorithm which reduces f by repeatedly subtracting multiples from the finite set of polynomials F. The crucial properties we will need is:

- $nf(f, F) \in \mathrm{Idl}(f, F)$,
- $f \in \mathrm{Idl}(nf(f, F), F)$,
- $nf(f, F) = 0 \Rightarrow f \in F$,
- $(nf(g, F) \neq 0 \ \&\ f \in \mathrm{Idl}(F) \ \&\ f \neq 0) \Rightarrow mdeg(f) \not| \ mdeg(nf(g, F))$.

This has a direct formal counterpart:

```
Variable nf : poly -> (list poly) -> poly.
Hypothesis nf_Cb : (f:poly)(F:(list poly)) (Cb (nf f F) (cons f F)).
Hypothesis Cb_nf : (f:poly)(F:(list poly))(Cb f (cons (nf f F) F)).
Hypothesis zerop_nf_cb: (F:(list poly)) (f:poly) (zerop (nf f F)) ->
                          (Cb f F).
Hypothesis get_is_correct:
           (g, f:poly)
           (F:(list poly)) ~ (zerop (nf g F)) -> (In f F) -> ~ (zerop f)
            -> ~ (tdiv (mdeg f) (mdeg (nf g F))).
```

Finally, we need a notion of Gröbner base. In the literature, F is a Gröbner base if for all g, if $g \in \mathrm{Idl}(F)$ then $nf(g,F) = 0$ regardless of the exact order of the reduction-strategy nf. Since nf is a computable function, and equality on polynomials is decidable, this gives a decision procedure for membership of those ideals for which one can compute a Gröbner base. However, with this definition it is not direct to check whether F is a Gröbner base, so one use a more technical definition using *S-polynomials*; an S-polynomial of two polynomial is essentially their critical pair in the sense of Knuth-Bendix completion [KB70] for the reduction algorithm nf.

In this terminology, when there are no more non-trivial critical pairs, we have a Gröbner base, and the naive Buchberger algorithm consists of adding such critical pairs until we have a Gröbner base. The termination of this process follows from Dickson's lemma. The main important property for us at this level of abstraction is the following:

> for any list of polynomials F, either F is a Gröbner base, or there exists a non-zero polynomial g in the ideal of F which is normalised with regards to F (this polynomial is usually a normalised S-polynomial).

For a naive formalisation, this is enough to get an algorithm; add the polynomials provided by the above property until we have a Gröbner base.

However, for the efficiency of the algorithm, it is crucial to have two arguments, first the actual argument, and then an argument where the result accumulates. This requires a technical relation $\cong$ (formally written `meq`) which relates lists of polynomials F to lists of critical polynomials of F which is used as invariant during the computation. For this relation, it is enough to require that for any list F there exists a list G such that $F \cong G$, and if $F \cong G$, then either F is a Gröbner base, or there exists a new set G' and a non-zero normalised polynomial $nf(g,F)$ which is in the ideal of F and for which $\{nf(g,F)\} \cup F \cong G'$

```
Variable Grobner : (list poly) -> Prop.
Variable meq : (list poly)->(list poly)->Prop.
Hypothesis start_meq : (F:(list poly)) { G:(list poly) | (meq F G) }.
Hypothesis genNext : (F,G:(list poly))(meq F G) ->
                     { G':(list poly) &
                         { g:poly | ((meq (addL poly (nf g F) F) G') /\
                                     ((Cb (nf g F) F) /\
                                      ~(zerop (nf g F)))) } }
                      + { (Grobner F) }.
```

Note that `{ G:(list poly) | (meq F G) }` is a special existential quantification claiming the existence of G, in *Set* since we want to compute it, with a proof of $F \cong G$ in *Prop*, since we don't want to compute this proof in the extracted program. Similarly, `{ a:A & P(a) }` is existential

quantification in *Set* with both `A` and `P(a)` in *Set*, and `+` is a special disjunction with constructor in *Set* and disjuncts in *Prop*, which will extract to a datatype isomorphic to the booleans.

These two hypotheses gives the abstract completion algorithm of F: we start with a G such that $F \cong G$, either F is a Gröbner base, or there exists a set G' and a polynomial g such that $\{nf(g,F)\} \cup F \cong G'$ and $\{nf(g,F)\} \cup F$ generates the same ideal as F. Either this is a Gröbner base, or we continue in same way, adding more normalised non-zero polynomials to F until it is completed.

The final problem is to prove that this process terminates, and to package this argument in an existential proof of the existence of Gröbner bases. To this end, we will use an inductive definition, very similar to the accessibility predicate, and rephrase and prove a version of Dickson's lemma [Dic13] for this definition.

2.2 Dickson's Lemma

In this subsection, we present a constructive proof of Dickson's lemma [Dic13] and show how this implies the constructive existence of Gröbner bases. The proof is a translation of a classical proof [CP99]. Dickson's lemma says that for any infinite sequence of n-tuples of natural numbers $\sigma_1, \sigma_2, \ldots$, there exists $i < j$ such that $\sigma_i \leq^n \sigma_j$, where $(a_1, \ldots, a_n) \leq^n (b_1, \ldots, b_n)$ if $\forall 0 < i \leq n.\, a_i \leq b_i$. We call a relation *well* if it satisfies this condition.

Remark 1 *The proofs in this section only require $<$ to be a decidable relation which is well–founded on its underlying set A, with $a \leq b$ defined as $\neg(b < a)$. However, we will only instantiate the theorems for $<$ being the less-than relation on $\mathbb{N}$.*

2.2.1 Inductive definition: Bar

We want to express in type theory, extended with inductive definitions, what it means for a relation R over a set B to be well. To this end, we define $Good_R(b_0 \cdots b_m)$ to be $\exists i < j \leq m.\, b_i \, R \, b_j$. We use an inductive definition of *bar* [ML68] to express that for any infinite sequence $b_0 b_1 \cdots$, $Good_R(\sigma)$ will eventually hold for an initial segment σ of $b_0 b_1 \cdots$.

Definition 2.1 *Given a set B and a predicate P over the lists of B, we define inductively when the predicate P* bars *σ, written $P \mid \sigma$:*

$$\frac{P(\sigma)}{P \mid \sigma} \qquad \frac{\forall b.\, P \mid \sigma.b}{P \mid \sigma}$$

This is a generalised inductive definition [Acz77], which comes with a transfinite induction principle:

$$\frac{\begin{array}{c}\forall \rho.\, P(\rho) \Rightarrow \Psi(\rho), \\ \forall \rho.\, (\forall b.\, P \mid \rho.b) \;\Rightarrow\; (\forall b.\, \Psi(\rho.b)) \;\Rightarrow\; \Psi(\rho)\end{array}}{P \mid \sigma \;\Rightarrow\; \Psi(\sigma)}$$

Here $\sigma.b$ is the list σ extended with the element b. Intuitively, $P \mid \sigma$ means that P will eventually hold for any extension of σ. Classically, assuming the axiom of dependent choices, $Good_R \mid []$ is provable iff R satisfies the classical definition of well. This justifies us to define R to be well iff $Good_R \mid []$ is provable.

```
Inductive ExistsL [A:Set;P:(Pred A)] : (list A) -> Prop :=
   FoundE : (a:A)(l:(list A))(P a) -> (ExistsL A P (cons a l))
 | SearchE : (a:A)(l:(list A))(ExistsL A P l) -> (ExistsL A P (cons a l)).
Inductive GoodR [A:Set; R:(Rel A)] : (list A) -> Prop :=
   FoundG : (a:A)(l:(list A))(ExistsL A ([x:A](R x a)) l) ->
            (GoodR A R (cons a l))
 | SearchG : (a:A)(l:(list A))(GoodR A R l) -> (GoodR A R (cons a l)).
Inductive Bar [A:Set;P:(list A)->Prop] : (list A)->Prop :=
     Base : (l:(list A))(P l) -> (Bar A P l)
 |   Ind  : (l:(list A))((a:A)(Bar A P (cons a l))) -> (Bar A P l).
Definition GRBar := [A:Set;R:(Rel A)] (Bar A (GoodR A R)).
Definition WR := [A:Set;R:(Rel A)] (GRBar A R (nil A)).
```

Lemma 2.2 *If* $\forall \sigma, \rho, \gamma.\, P(\sigma\rho) \Rightarrow P(\sigma\gamma\rho)$*, then* $\forall \sigma, \rho, \gamma.\, P \mid \sigma\rho \;\Rightarrow\; P \mid \sigma\gamma\rho$.

Proof 1 *Immediate by induction on the proof of* $P \mid \sigma\rho$.

2.2.2 Open Induction

Given two relations R and S over sets A and B respectively, the product relation, $R \times S$, over $A \times B$ is defined as $(a, b)\,(R \times S)\,(a', b') = a\,R\,a'\ \&\ b\,S\,b'$. Following [Coq92], we define a predicate $M(\sigma)$, expressing that an initial sequence σ of pairs in $A \times B$ is *minimal w.r.t.* $<$, by recursion on σ:

$$\begin{aligned} M([\,]) &= \top, \\ M(\sigma.(x,b)) &= M(\sigma)\ \&\ \forall y.\, y < x \Rightarrow \forall b.\, Good_{\leq \times R} \mid \sigma.(y,b). \end{aligned}$$

The predicate M plays the rôle of a minimal bad sequence in the following sense: if $M(\sigma)$ holds, and ρ has a lexicographically smaller sequence of first components, then $Good_{\leq \times R} \mid \rho$ should hold.

Raoult's open induction principle can be expressed using these definitions:

Theorem 2.3 (Open Induction) *For any finite sequence* σ *of pairs in* $A \times B$*, if* $M(\sigma)$ *and* $\forall a, b.\, M(\sigma.(a,b)) \;\Rightarrow\; Good_{\leq \times R} \mid \sigma.(a,b)$*, then* $Good_{\leq \times R} \mid \sigma$.

Proof 2 *Assume* $M(\sigma)$ *and* $\forall a, b.\, M(\sigma.(a,b)) \;\Rightarrow\; Good_{\leq \times R} \mid \sigma.(a,b)$ *holds. By induction on* x*, we prove* $\forall x.\, \forall b.\, Good_{\leq \times R} \mid \sigma.(x,b)$*: Assume* $\forall y.\, y < x \Rightarrow \forall b.\, Good_{\leq \times R} \mid \sigma.(y,b)$*. From this we directly obtain* $M(\sigma.(x,b))$*, and by hypothesis,* $Good_{\leq \times R} \mid \sigma.(x,b)$.

This theorem is a simplification of that in [Coq92] but it uses only generalised inductive definitions iterated once [Acz77]. It will interpret the argument: if $Good_{\leq \times R} \mid \sigma$ holds under the assumption that σ starts a minimal bad sequence, then $Good_{\leq \times R} \mid \sigma$ holds without this assumption as well.

```
Section OpenIndGoodRel.

Variable A:Set.
Variable lt,R:(Rel A).
Variable wflt:(well_founded A lt).
```

```
Inductive Min : (list A)->Set :=
   nmin : (Min (nil A))
 | cmin : (a:A)(l:(list A))(Min l)->((y:A)(lt y a) ->
           (GRBar A R (cons y l)))->(Min (cons a l)).

Lemma OpenInd : (xs:(list A))(Min xs)->((a:A)(Min (cons a xs))->
                 (GRBar A R (cons a xs)))->(GRBar A R xs).
 ...
End OpenIndGoodRel.
```

Now, a classical minimal bad sequence argument proof of Dickson's lemma can be interpreted as:

Lemma 2.4 *If* $Good_R \mid b_1 \cdots b_m$ *holds, then*

$$\forall x_1, \ldots, x_m.\, M((x_1, b_1) \cdots (x_m, b_m)) \;\Rightarrow\; Good_{\leq \times R} \mid (x_1, b_1) \cdots (x_m, b_m).$$

Proof 3 *By induction on the proof of* $Good_R \mid b_1 \cdots b_m$*:*

$Good_R(b_1 \cdots b_m)$**:** *Then there exists a* $i < j \leq m$ *such that* $b_i \, R \, b_j$*. Now, by cases on the decidable relation* $<$*, we have either* $\neg(x_i > x_j)$*, that means* $x_i \leq x_j$ *so* $(x_i, b_i)\,(\leq \times R)\,(x_j, b_j)$ *holds, hence* $Good_{\leq \times R}((x_1, b_1) \cdots (x_m, b_m))$*. In the other case we have* $x_i > x_j$*, and from* $M((x_1, b_1) \cdots (x_m, b_m))$ *we directly obtain* $M((x_1, b_1) \cdots (x_i, b_i))$*, which entails* $Good_{\leq \times R} \mid (x_1, b_1) \cdots (x_{i-1}, b_{i-1})(x_j, b_j)$*, and by Lemma 2.2, we are done.*

$\forall b.\, Good_R \mid (b_1 \cdots b_m b)$**:** *Immediate by Theorem 2.3 and IH.*

Corollary 2.5 (Dickson's lemma) *For all* $n \in \mathbb{N}$*,* $Good_{\leq^n} \mid [\,]$*.*

Proof 4 *By induction on* n*: the case* $n = 0$ *is trivial. Otherwise,* $n = m + 1$*, and* $Good_{\leq^m} \mid [\,]$ *holds. We instantiate the development above with* R *being* $\leq^m$*, and by Lemma 2.4 we are done.*

```
Section Dickson.

Variable A,B:Set.
Variable lt:(Rel A).
Variable R:(Rel B).
Variable wfgt:(well_founded A lt).
Variable declt:(DecRel A lt).
Variable wR:(WR B R).

Definition leq := [a,b:A] ~(lt b a).
Definition GBarlR := (GRBar (A*B) (ProdRel A B leq R)).
Lemma keylem : (bs:(list B))(GRBar B R bs)->((us:(list (A*B)))
                 (bs = (sndL us))->(MinD us)->(GBarlR us)).
 ...
End Dickson.

Lemma dicksonL : (n:nat)(WR (mon n) (mdiv n)).
```

2.2.3 Non-informative Bar

There is a technical problem with the above definition of Bar. During computation, one would ideally not compute proofs of bar unless it is really needed. An example where it is needed is the proof of Higman's lemma for undecidable relations [Fri97]; in that case the proof of Bar is used in a similar way as a proof of decidability.

However, in the case of Buchberger's algorithm, it seems clear that the proof of Bar is not needed during computation (it might help in optimisation of the program however). The problem is to make sure that the proof of Bar is not used in the computation, and hence removed during extraction. In Coq, this is solved by the type-system by a distinction between *Set* and *Prop*; elements in *Set* are datatypes that may be used in computation, whereas elements in *Prop* are not. This means that one cannot compute (e.g. use elimination rules) with elements of types in *Prop* to produce elements of types in *Set*.

The first problem is that the existence of Gröbner bases is proved by induction on a proof of Bar, so Bar cannot be in *Prop* and removed during extraction. But in Coq there is a trick based on the following observation: inductive datatypes with only one constructor is *non-informative* in the sense that they cannot affect computations. This means that Coq allows e.g. the accessibility predicate to be in *Prop* even if it is used to compute elements in *Set*. However, in the case of two constructors, as the case of Bar above, this trick does not work.

Fortunately, there exists an alternative inductive definition of Bar, with only one constructor [JL91]:

Definition 2.6 *Given a set B and a predicate P over the lists of B, we define inductively $P \parallel \sigma$:*

$$\frac{\forall m.\, \neg P(m_1, \ldots, m_k, m) \Longrightarrow P \parallel (m_1, \ldots, m_k, m)}{P \parallel (m_1, \ldots, m_k)}$$

```
Inductive BarML [A:Set;P:(list A)->Prop] : (list A)->Prop :=
 | IndML  : (l:(list A))((a:A)~(P (cons a l))->
              (BarML A P (cons a l))) -> (BarML A P l).
```

This definition has advantages in describing the notion of *Noetherian* (by letting $P(a_1, \ldots, a_k, a_{k+1}) = a_{k+1} \notin \mathrm{Idl}(a_1, \ldots, a_k)$), for example it is provable that implementations of the reals form a noetherian ring with this definition, whereas with the old definition it is not possible. Note that this can be seen as an accessibility predicate, and $P \parallel ()$ essentially says that the relation $\vec{a} \succ \vec{b}$ iff $\vec{b} = (\vec{a}, a)$ and $\neg P(\vec{a}, a)$, is well-founded.

Now, it is direct to prove $P \mid (m_1, \ldots, m_k) \Longrightarrow P \parallel (m_1, \ldots, m_k)$ by induction on the proof of $P \mid (m_1, \ldots, m_k)$. The opposite direction seems to require more assumptions on P, for example decidability of P.

```
Lemma Bar2BarML : (A:Set)(R:(Rel A))(l:(list A))
                     (GRBar A R l) ->(GRBarML A R l).
```

This implies that Dickson's lemma holds for this formulation as well. We will now use this definition of Bar to prove the existence of Gröbner bases.

In fact, in Théry's development, he used the fact that the relation $\vec{a} \succ \vec{b}$ iff $\vec{b} = (\vec{a}, nf(a, \vec{a}))$ and $nf(a, \vec{a}) \neq 0$ is well-founded, but he only gave a classical proof of this fact using a classical formulation of Dickson's lemma. We can now directly prove that this relation is well–founded, thus making his external development completely constructive.

2.3 Constructive Existence of Gröbner Bases

Following Buchberger's algorithm we can use Dickson's lemma to prove the existence of Gröbner bases for any finitely generated (f.g.) ideal in $K[X_1, \ldots, X_m]$:

Theorem 2.7 *Every f.g. ideal* $F = \mathrm{Idl}(f_1, \ldots, f_n)$ *has a Gröbner basis* G.

Proof 5 *We prove this using Dickson's lemma. Let* $m_i = \mathrm{mdeg}(f_i)$. *Define* $Bad(\sigma)$ *as* $\neg Good(\sigma)$. *We can assume that* $Bad(m_1 \cdots m_n)$ *holds; if* $m_i \leq m_j$ *for* $i < j$, *we repeatedly reduce* F *by dividing* f_j *by* f_i. *The result follows from Dickson's lemma and the following lemma:*

$$Good_{\leq} \mid\mid m_1 \cdots m_k \;\Rightarrow\; \forall f_1, \ldots, f_k.\, (\forall i.\, m_i = \mathrm{mdeg}(f_i)) \;\Rightarrow\; Bad(m_1 \cdots m_k) \;\Rightarrow\; \exists G.\, G \text{ is a Gröbner basis for } f_1, \ldots, f_k,$$

which is proved by induction on the proof of $Good_{\leq} \mid\mid m_1 \cdots m_k$:

$\forall m.\, Bad(m_1 \cdots m_k m) \Rightarrow Good_{\leq} \mid m_1 \cdots m_k m$**:** *Assume* $f_1, \ldots, f_k$ *with* $\mathrm{mdeg}(f_i) = m_i$ *for all* i *and* $Bad(m_1 \cdots m_k)$. *By the Gröbner base criteria, either* $\{f_1, \ldots, f_k\}$ *is a Gröbner base, in which case we are done, or there exists a polynomial* f_{k+1} *which is normalised with regards to* F. *Then* $m_l \not\leq \mathrm{mdeg}(f_{k+1})$ *for all* $l \leq k$, *so if* $Bad(m_1 \cdots m_k)$, *then* $Bad(m_1 \cdots m_k \mathrm{mdeg}(f_{k+1}))$. *Hence, by IH, we have a Gröbner basis for* $F \cup \{f_{k+1}\}$, *and since* f_{k+1} *is in* $\mathrm{Idl}(F)$, *this is a Gröbner basis for* F.

This is an integrated version of Buchberger's algorithm: while F is not a Gröbner basis, add normalised S-polynomials to F. To get an efficient algorithm, we reformulate the above lemma and proof to use an extra argument, related by $\cong$. Optimisations of the algorithm can be made by changing the way normalised polynomials are found in the criteria for Gröbner bases (`genNext`). Formally, we can prove:

```
Lemma lemGB : (M:(list trm))(GRBarML trm tdiv M) -> (Bad M) ->
                ((F,G:(list poly))(meq F G)->
                  ((g:poly)(In g F)-> ~(zerop g))->
                   ((rev (mdegL F)) = M) ->
                { G:(list poly) | (eqI F G) /\ (Grobner G) }).
```

Note that this lemma uses the reverse function on lists, since in the completion process, we want to add polynomials to the end of the list, and the recursion principle for `Bar` only supports adding polynomials to the front of the list. Also note that this algorithm is very general in its abstract formulation, and captures a number of optimized versions of Buchberger's algorithm.

```
Lemma exbad_rev : (F:(list poly))
                    { G:(list poly) | (eqI F G) /\
                                      ((Bad (rev (mdegL G))) /\
                                       ((f:poly)(In f G)->~(zerop f))) }.

Theorem exGB : (F:(list poly))
                { G:(list poly) | (eqI F G) /\ (Grobner G)}.
```

3 Conclusions

The formalisation has really been a combination of two previous formalisations in two different systems. As such, it is a very positive experience that it only took one month to learn Coq and combine the two formalisations. Another positive experience is that 90 percent of Thèry's formalisation could be reused. This was much due to the use of abstractions in Coq, which made it possible to keep the development in small modules which could quite quickly be type-checked. As reported in [Thé98] however, the use of abstraction in Coq was not ideal, and we will compare it to the possibilities for abstractions in Agda below.

A problem with the integrated approach in Coq is that one have to specify exactly what should be kept in the extraction process (by choosing between the types *Set* and *Prop*). This is expected since it is non-trivial to automatically deduce this information. The problem with this is that several proofs need to be changed in order not to use case-analysis on sets which should live in *Prop*, which can be somewhat cumbersome. Furthermore, it took some time and thinking before the right combination of *Set* and *Prop* was found.

An open problem is to modify the proof in [CP99] to obtain a short constructive proof of Hilbert's basis theorem, using the one-constructor definition of Noetherian. In [JL91], Jacobsson and Löfwall obtained such a proof in a very general form, but their proof seems quite involved to make a quick formalisation. This is essential for being able to extract programs which implements more general versions of Buchberger's algorithm.

3.1 Agda vs. Coq

Since we now have two quite similar formalisation in the two different systems Agda and Coq, we would like to compare the two systems.

3.1.1 Abstractions

One interesting point is the support for abstractions, which was essential for making this formalisation short. Essentially, abstractions makes formalisations shorter by replacing complicated expressions by variables, which later can be instantiated by these complicated expressions. This makes the formalisation short, more readable, and simpler to change. Agda and Coq provides different mechanisms for abstractions.

In Agda, a typical abstraction may look something like this:

```
package pk1 (A,B::Set)
 f1 :: (a1,a2::A) A = ...
 f2 :: (a::A)(b::B) B = ...
```

This defines two functions under the common assumption of two sets A and B. Now, to use these functions, we first instantiate the package pk1 with to sets, and projects the two components using dot-notation:

```
package mypk = pk1 Nat Bool
mypk.f1 0 0
mypk.f2 0 True
```

alternatively, using the `open` construct, the functions can be put directly in the context:

```
package mypk = pk1 Nat Bool
open mypk use f1, f2
 f1 0 0
 f2 0 True
```

In Coq, the same development would look something like:

```
Section pk1.
Variable A,B:Set.
 Definition f1 : (a1,a2:A) A := ...
 Definition f2 : (a:A)(b:B) B := ...
End pk1.
```

The difference lies in how the sets A and B are abstracted; Coq directly translates the above development to something corresponding to:

```
 Definition f1 : (A:Set)(a1,a2:A) A := ...
 Definition f2 : (A,B:Set)(a:A)(b:B) B := ...
```

Note that `f1` is only parametrised on the set A. This is clearly a weaker form of abstraction; the instantiation has to be done for each function, and each time the function is used. A more annoying problem is that not all functions depend on all the variables abstracted over; in the instantiation of `f1` we have to know that it does not use B. This is a big nuisance for big formalisations with many assumptions. For example, Théry's formalisation contains theorems depending on more than 50 assumptions taken from a context of many more assumptions, and the user have to sort out which assumptions the theorems depend on, and in which order. Needless to say, this treatment also leads to less efficient type-checking, since terms are instantiated over and over again.

To summarize the advantages of Agda's abstraction mechanism:

- abstraction of a common context for a *collection* of definitions,
- simultaneous instantiation of a common context for such a collection of definitions.

3.1.2 Proof-scripting vs. Proof-editing

Agda and Coq differs in the style in which a proof is built. In Agda, the user writes the proof-term herself, whereas in Coq the user writes scripts which when executed builds up the proof term. This means that the Coq-user need not concern herself with the proof-term. This might be an advantage when the proof-term is not interesting and the proof is big (tactics can be used to quickly produce a big proof-term), but might be an dissadvantage when the proof-term i relevant (the user will directly see the proof-term).

In our part of this formalisation, we did not need to do any big proof without interesting proof-term, so it is hard to judge Coq's advantage on this point. However, in the complete development with the instantiations, there were big proofs with no interesting proof-term, such as the properties of polynomial ring, that could be made by tactics. But in the current state of Coq, there were not very much tactical support for making these proofs substantially shorter than the Agda-proof would have been.

3.2 Proof optimisation in Coq

In the comparison between the two programs extracted from the two different formalisations, it is important to remember that the extraction process did not attempt any optimisations that used the proof. It may be likely, by the results of Goad [Goa80a, Goa80b], that a more clever extraction procedure which performs proof optimisation first, should extract a more efficient program from the integrated formalisation. Unfortunately, there exist no such proof optimisation in Coq.

We believe the integrated approach in general, and this work in particular, motivates such a proof optimiser in Coq. As shown in this work, the integrated approach requires not too much extra development; especially when compared to the constructive external approach.

Acknowledgement

I would like to thank the Lemme group at INRIA/Sophia-Antipolis for providing a very friendly and stimulating environment, and for all their help in using Coq. I would especially like to thank Laurent Théry for all the discussions and explanations during this work. This work was funded by CFC. I also would like to thank Chalmers University for providing funding and resources for the last weeks when finishing this work.

This report was completed in March 2000.

References

[Acz77] P. Aczel. An Introduction to Inductive Definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland Publishing Company, 1977.

[Aug98] L. Augustsson. Cayenne - a language with dependent types. Technical report, Department of Computing Science, Chalmers University of Technology, 1998. Homepage: http://www.cs.chalmers.se/~augustss/cayenne/.

[Buc65] B. Buchberger. *An Algorithm for Finding a Basis for the Residue Class Ring of a Zero-Dimensional Polynomial Ideal (German).* PhD thesis, University of Innsbruck, 1965.

[Buc85] B. Buchberger. Gröbner bases: An algorithmic method in polynomial ideal theory. In N. K. Bose, editor, *Multidimensional systems theory*, pages 184–232. Reidel Publ. Co., 1985.

[Buc98] B. Buchberger. Introduction to Gröbner bases. In B. Buchberger and F. Winkler, editors, *Gröbner bases and applications*, pages 3–31. Cambridge University Press, 1998.

[BW93] T. Becker and V. Weispfenning. *Gröbner bases*, volume 141 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 1993. In cooperation with H. Kredel.

[CLO97] D. Cox, J. Little, and D. O'Shea. *Ideals, varieties, and algorithms.* Undergraduate Texts in Mathematics. Springer-Verlag, New York, second edition, 1997.

[Coq92] Th. Coquand. Constructive topology and combinatorics. In *proceeding of the conference Constructivity in Computer Science, San Antonio, LNCS 613*, pages 28–32, 1992.

[Coq96] Th. Coquand. Computational content of classical logic. In A. M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation.* Cambridge University Press, 1996.

[Coq98] C. Coquand. The homepage of the Agda type checker. Homepage: http://www.cs.chalmers.se/~ catarina/Agda/, 1998.

[CP99] Th. Coquand and H. Persson. Gröbner bases in type theory. In *Selected papers from Types'98, LNCS 1657*, pages pp 33–46. Springer-Verlag, 1999.

[Dic13] L. E. Dickson. Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. *Am. J. Math.*, 35:413–422, 1913.

[Dyb90] P. Dybjer. Comparing integrated and external logics of functional programs. *Science of Computer Programming*, 14:59–79, 1990.

[Fri97] D. Fridlender. *Higman's Lemma in Type Theory*. PhD thesis, Chalmers University of Technology and University of Göteborg, Sweden, 1997.

[Frö97] R. Fröberg. *An introduction to Gröbner bases*. John Wiley & Sons, 1997.

[Gir86] J-Y Girard. Linear logic and parallelism. In M. Venturini Zilli, editor, *Mathematical Models for the Semantics of Parallelism*, number LNCS 280, pages 166–182. Springer-Verlag, September 1986.

[Goa80a] C. Goad. *Computational Uses of the Manipulation of Formal Proofs*. PhD thesis, Computer Science Department, Stanford University, August 1980.

[Goa80b] C. Goad. Proofs as Descriptions of Computation. In *Proceedings of the 5th Conference on Automated Deduction*, volume 87 of *Lecture Notes in Computer Science*, pages 39–52. Les Arcs, France, Springer-Verlag, 1980.

[Hal98] T. Hallgren. Home Page of the Proof Editor Alfa. `http://www.cs.chalmers.se/~hallgren/Alfa/`, 1998.

[HKPM97] G. Huet, G. Kahn, and C. Paulin-Mohring. The Coq proof assistant: A tutorial. Technical report, Rapport Technique 204, INRIA, 1997.

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, London, 1980.

[HT98] J. Harrison and L. Théry. A sceptic's approach to combining HOL and Maple. *Journal of Automated Reasoning*, pages 279–294, 1998.

[JL91] C. Jacobsson and C. Löfwall. Standard bases for general coefficient rings and a new constructive proof of Hilbert's basis theorem. *J. Symbolic Comput.*, 12(3):337–371, 1991.

[KB70] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In *Computational Problems in Abstract Algebra (Proc. Conf., Oxford, 1967)*, pages 263–297. Pergamon, Oxford, 1970.

[Ler99] X. Leroy. The Objective CAML system release 2.04. Homepage: http://caml.inria.fr, 1999.

[ML68] P. Martin-Löf. *Notes on Constructive Mathematics*. Almqvist & Wiksell, 1968.

[PH96] J. Peterson and K. Hammond. The Haskell 1.3 Report. Technical Report YALEU/DCS/RR-1106, Yale University, 1996.

[PT] L. Pottier and L. Théry. Certified gröbner bases computations. Homepage: http://www-sop.inria.fr/croap/CFC/Gbcoq.html.

[Thé98] L. Théry. Proving and computing: A certified version of the Buchberger's algorithm. In *Proceeding of the 15th International Conference on Automated Deduction, Lindau, Germany, LNAI 1421*, 1998.

[typ98] Calculemus and types '98 conference, electronic proceedings. Homepage: http://www.win.tue.nl/math/dw/pp/calc/, 1998.

[WB81] W. Pohlers W. Sieg W. Buchholz, S. Feferman. *Iterated inductive definitions and subsystems of analysis*, volume Recent Proof-Theoretical Studies of *Lecture Notes in Mathematics 897*. Springer-Verlag, 1981.

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

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

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