Automated Simplification and Deduction for Engineering Formulas

John J Wavrik
Dept. of Math
Univ. of Calif. - San Diego

Abstract

The commutative version of the Gröbner Basis Algorithm has become one of the most powerful tools in computer algebra. The recent adaptation of the algorithm to non-commuting variables has potential applications to matrix and operator expressions. A Forth-based research system was used to implement and study applications of a non-commutative variant of the Gröbner Basis Algorithm. It has been used in automated simplification and deduction for formulas in engineering. The paper will discuss this work and the use of Forth in producing systems for mathematics research.

Introduction

The Gröbner Basis Algorithm for polynomials in commuting variables was developed in the mid 1960s. This algorithm has revolutionized the field of computational algebraic geometry in the past decade. It also has become an important tool in solving systems of algebraic equations and is now used for this purpose in all modern computer algebra systems.

An adaptation of the Gröbner Basis Algorithm to polynomials in non-commuting variables was made in the late 1980's. A Forth-based research system was created by the author to implement and this algorithm and its applications. In joint work with Prof. J William Helton, this system was used to study applications to simplification and deduction for matrix and operator expressions which arise in engineering.

This paper is in three parts. Part I contains background on the division algorithm for several variables and Gröbner bases. Part II discusses the application to simplification and deduction. Part III contains information about the use of Forth in the research system used for this project.

Part I - Division and Gröbner Bases

The Gröbner Basis algorithm arises in connection with a reduction process which generalizes the well-known "long division" algorithm to polynomials in several variables. The standard algorithm divides a polynomial f by a polynomial g to yield a quotient q and remainder r. We have f = q*g + r. The degree of r is less than the degree of g.

In this algorithm we use the natural ordering of polynomials by degree. The terms of g are in descending order and the leading term LT(g) is used to determine the partial quotient at each step. The process continues until we obtain a polynomial whose terms are not divisible by LT(g). This is the remainder, r.

The algorithm provides a test for deciding if a polynomial f is or is not a multiple of g: f is a multiple of g if and only if r = 0.

$$
\begin{array}{r}
2x-1 \\
x+1\,\overline{)\,2x^2+x+2} \\
2x^2+2x \\
\hline
-x+2 \\
-x-1 \\
\hline
3
\end{array}
$$

The natural generalization to polynomials in several variables $x_1,..,x_n$ is to "divide" a polynomial f by a set of polynomials G. The goal is to find an algorithm for writing $f = a_1g_1 + ... + a_mg_m + r$ (in the commutative case)[1] where r is, in some sense, the remainder.

There is no natural ordering on the monomials in several variables -- but an ordering can be chosen from among several possibilities. In what follows we will assume a fixed order[2] so that every polynomial f has a leading term, LT(f), which is largest in this ordering. The essence of the division algorithm is to systematically eliminate terms in which any $LT(g_i)$ occurs, replacing them by a sum of lower order terms.

Division Algorithm

Input: polynomials f, $g_1,...,g_m$
Output: r so that (1) $f = LinComb(g_1,...,g_m) + r$
(2) no term of r is divisible by any $LT(g_i)$

```
r := f
while ∃ i, t  s.t. Divides(LT(gi),t)     ( t a term of r)
        write t = c L·LT(gi)·R
        r := r - c L·gi·R
```

The algorithm stops because each term of r is replaced by terms of lower order and we use orderings for which there are no infinite descending sequences. It is easy to modify this algorithm to keep account of the linear combination of g_i, but is really r that is of importance.[3]

Notice that the algorithm, as given, depends on arbitrary choices: the order in which the $LT(g_i)$ are used to test for divisibility, the order in which the terms of r are examined for divisibility, and which occurrence of $LT(g_i)$ in t is used, if it occurs more than once.

EXAMPLE 1

$f = x^2y$, $g_1 = xy - 1$, $g_2 = x^2 - y$
We first reduce by g_1, subtract $f - xg_1$ to obtain $r = x$
We first reduce by g_2, subtract $f - yg_2$ to obtain $r = y^2$

⌘

[1] In the non-commutative case, we write $f = \sum a_i L_i g_i R_i + r$ where the L_i and R_i are words in the variables, $g_i \in G$, and the sum is finite.

[2] The ordering on monomials is required to satisfy (1) $1 < M$ (2) $M_1 < M_2 \Rightarrow M \cdot M_1 < M \cdot M_2$ and $M_1 \cdot M < M_2 \cdot M$ (3) $<$ is a well-ordering (no infinite descending chains)

[3] The algorithm is stated for a finite set of G, but it can be applied to an infinite set provided that the decision Divides(LT(g),t) can be made in finite time.

In each case the process stops because we obtain an r which is not divisible by any $LT(g_i)$.

EXAMPLE 2

With g_1 and g_2 as above, the polynomial $y^2 - x$ can be written as a linear combination of g_1 and g_2: $y^2 - x = x\, g_1 - g_2\, y$. However if we use the division algorithm we get $r = y^2 - x$ which is not zero.

There is, therefore, a natural and simple way to generalize the division algorithm to polynomials in several variables. Unfortunately the division process is badly behaved in general. As the first example shows, one can, in fact, get different remainders depending on the choices made in carrying out the algorithm. If r=0 then f is a linear combination of the g_i. The second example shows that the converse need not be true.

Gröbner Bases

A set, I, of polynomials is called an ideal if for all $f,g \in I$ and all polynomials p we have f + g, pf, and $fp \in I$. The set of all linear combinations of $G = \{g_i\}$ is an ideal (called the ideal generated by G). G is called a basis of this ideal.

G is called a Gröbner Basis for the ideal, I, it generates if the division algorithm is nicely behaved in the sense that $\forall f, f \in I \Leftrightarrow r=0$[4]. It can be shown that, in this case, r is independent of the choices -- so it is a canonical form (aka normal form) for f and is denoted N(f,G).

The main theoretical results in the commutative case were obtained in the mid 1960's. Hironaka ([Hiro], 1964) proved the existence of a Gröbner Basis for any ideal. Buchberger ([Buch1], 1965) showed this independently. He also provided an algorithm for obtaining a Gröbner Basis from an arbitrary basis. By 1985, Buchberger [Buch2] substantially improved the efficiency of the algorithm. He and others also explored a variety of applications.

In the non-commutative case, ideals may have infinite bases. In 1986 Mora [Mora] provided an algorithm which, if it terminates, gives a finite Gröbner Basis and conditions under which an infinite basis satisfies the Gröbner property.

The commutative case of the Gröbner Basis algorithm has been a breakthrough in computational algebra. By providing a computational test for membership in an ideal

⌘———————————————

[4] In view of the fact that r is not unique, it may be clearer to say G is not a Gröbner basis if we can find an $f \in I$ which has a non-zero remainder for some choice of reduction.

(among other things) it provides a tool for computing in quotient rings and working out details of explicit examples. It has also been used in solving systems of equations: for some term orderings, the Gröbner Basis yields an equivalent system of equations in which the variables are successively eliminated (solving systems of linear equations by Gauss elimination is a special case).

Part II - Simplification and Deduction

SIMPLIFICATION:

The division algorithm can also be thought of as the repeated application of rewrite rules derived from the polynomials g_i. The process of multiplying $y^2 - x + 1$ by suitable factors and subtracting is equivalent to the process of scanning the terms of f for any occurrence of y^2 and then making the replacement $y^2 \rightarrow x - 1$. Thus a divisor, g, gives rise to a rewrite rule LHS $\rightarrow$ RHS where LHS = LT(g) and RHS = LT(g) - g. LHS is replaced, where it occurs, by a sum of terms of lower order.

After I gave a talk about Gröbner Bases a colleague, Bill Helton, asked if this technology would be useful in simplifying complicated formulas which arise in engineering. He supplied some examples. Here is one:

```
coexz = -tp[C1] ** C1 - XX ** B2 ** inv[d12] ** C1 - tp[C1] ** inv[tp[d12]] ** tp[B2] **
XX - XX ** inv[Id - YY ** XX] ** B1 ** inv[d21] ** C2 - XX ** inv[Id - YY ** XX] ** B1 **
tp[B1] ** XX + inv[YY] ** inv[Id - YY ** XX] ** B1 ** inv[d21] ** C2 + inv[YY] ** inv[Id -
YY ** XX] ** B1 ** tp[B1] ** XX - XX ** B2 ** inv[d12] ** inv[tp[d12]] ** tp[B2] ** XX -
XX ** inv[Id - YY ** XX] ** YY ** tp[C2] ** inv[tp[d21]] ** C2 - XX ** inv[Id - YY ** XX]
** YY ** tp[C2] ** inv[tp[d21]] ** tp[B1] ** XX + inv[YY] ** inv[Id - YY ** XX] ** YY **
tp[C2] ** inv[tp[d21]] ** inv[d21] ** C2 + inv[YY] ** inv[Id - YY ** XX] ** YY ** tp[C2] **
inv[tp[d21]] ** tp[B1] ** XX
```

tp denotes transpose, inv denotes inverse, C1, B2, etc are operators. Notice that this expression is a polynomial in variables like C1, inv[d12], etc. The only obvious relations were those among variables which represented subexpressions and their inverses (for example, d12 ** inv[d12] - Id). Thus I took for G a set of polynomials which represented obvious identities. A Gröbner Basis, G', was produced and the above expression was reduced by G'.

```
coexzsimp = YY ** (-tp[C1] ** C1 - XX ** B2 ** inv[d12] ** C1 - tp[C1] ** inv[tp[d12]] **
tp[B2] ** XX + tp[C2] ** inv[tp[d21]] ** inv[d21] ** C2 + tp[C2] ** inv[tp[d21]] ** tp[B1]
** XX - XX ** B2 ** inv[d12] ** inv[tp[d12]] ** tp[B2] ** XX) + B1 ** inv[d21] ** C2 + B1
** tp[B1] ** XX
```

Our success on these initial examples led to a systematic investigation ([HW], [HSW1], [HSW2]).

Automatic simplification by reduction is actually close to the way expressions are simplified "by hand". A human would scan an expression looking for ways to apply matrix identities. The most obvious thing to look for is an operator next to its inverse.

$$1 - x^{-1}x + x^2 \text{ simplifies to } x^2$$

An expert can bring to bear a large repertoire of identities.

$$1 - x^{-1}(1-xy)^{-1}x \text{ simplifies to } 1 - (1-yx)^{-1}$$

using the identity $(1-xy)^{-1}x = x(1-yx)^{-1}$.

We examined "models" consisting of a collection of simple expressions in one or more matrix variables.

EXAMPLE 3

> The simplest model, RESOL, uses expressions which are polynomials in x, x^{-1}, and $(1-x)^{-1}$. These three are taken as non-commuting variables in a polynomial ring with three variables. We may assign a=x, b=x^{-1}, c=$(1-x)^{-1}$. We choose the starting basis, G, to express obvious relations on a,b,c: G={ ba - 1, ab - 1, ca - c + 1, ac - c + 1}[5].

If the starting basis, G = {g_i } consists of matrix identities, then any linear combination of the g_i is a matrix identity. In particular, the Gröbner Basis, G', obtained from G will consist of are matrix identities. Moreover, N(f,G') will be equivalent to f since it differs from f by a matrix identity. In our models, the Gröbner Basis extended the initial set of obvious identities by a collection of more sophisticated identities.

Monomials are ordered first by word length and, for words of the same length, by lexicographic order. The simple expressions constituting the model were assigned to polynomial variables so that expressions which subjectively seem simpler were assigned earlier letters than those which seemed more complicated. The example above illustrates this for RESOL. Reduction is a simplification process: it replaces terms by terms lower in order. It also reduces equivalent terms to the same form -- which makes it possible to combine or cancel equivalent terms. The ordering has been chosen so that reduction will tend to eliminate the more complex basic expressions in favor of the simpler ones.

INFINITE BASES:

For all the models in our study, we obtained a Gröbner Basis which was either finite or a finite collection of "special" relations together with several infinite parametrized families.

⌘———————————

[5] For clarity, in what follows, we will write the corresponding matrix expression as the variable. The context will establish whether we are talking about x^{-1} as an independent variable in a polynomial ring, or as the inverse of the matrix x.

EXAMPLE 4

Here is the Reduced[6] Gröbner Basis obtained for expressions in

$$x < y < x^{-1} < y^{-1} < (1-x)^{-1} < (1-y)^{-1} < (1-xy)^{-1} < (1-yx)^{-1}.$$

The 12 starting relations were just those that come from the definition of inverse.

SPECIAL RELATIONS (starting relations marked with *)[7]

1*	$xx^{-1} - 1$	13*	$(1-y)^{-1}y - (1-y)^{-1} + 1$
2*	$x(1-x)^{-1} - (1-x)^{-1} + 1$	14	$(1-y)^{-1}y^{-1} - (1-y)^{-1} - y^{-1}$
3*	$yy^{-1} - 1$	15	$(1-xy)^{-1}x - x(1-yx)^{-1}$
4*	$y(1-y)^{-1} - (1-y)^{-1} + 1$	16	$(1-xy)^{-1}y^{-1} - x(1-yx)^{-1} - y^{-1}$
5*	$x^{-1}x - 1$	17	$(1-yx)^{-1}y - y(1-xy)^{-1}$
6	$x^{-1}(1-x)^{-1} - (1-x)^{-1} - x^{-1}$	18	$(1-yx)^{-1}x^{-1} - y(1-xy)^{-1} - x^{-1}$
7	$x^{-1}(1-xy)^{-1} - y(1-xy)^{-1} - x^{-1}$	19*	$xy(1-xy)^{-1} - (1-xy)^{-1} + 1$
8*	$y^{-1}y - 1$	20*	$yx(1-yx)^{-1} - (1-yx)^{-1} + 1$
9	$y^{-1}(1-y)^{-1} - (1-y)^{-1} - y^{-1}$	21	$(1-x)^{-1}y(1-xy)^{-1} - (1-x)^{-1}(1-xy)^{-1} - y(1-xy)^{-1} + (1-x)^{-1}$
10	$y^{-1}(1-yx)^{-1} - x(1-yx)^{-1} - y^{-1}$		
11*	$(1-x)^{-1}x - (1-x)^{-1} + 1$	22	$(1-y)^{-1}x(1-yx)^{-1} - (1-y)^{-1}(1- yx)^{-1} - x(1-yx)^{-1} + (1-y)^{-1}$
12	$(1-x)^{-1}x^{-1} - (1-x)^{-1} - x^{-1}$		

GENERAL RELATIONS

I $x(1-yx)^{-n}(1-x)^{-1} - (1-xy)^{-n}(1-x)^{-1} + (1-xy)^{-n}$

II $x(1-yx)^{-n}(1-y)^{-1} - (1-xy)^{-n}(1-y)^{-1} - x(1-yx)^{-n} + (1-xy)^{-(n-1)}(1-y)^{-1}$

III $y(1-xy)^{-n}(1-x)^{-1} - (1-yx)^{-n}(1-x)^{-1} - y(1-xy)^{-n} + (1-yx)^{-(n-1)}(1-x)^{-1}$

IV $y(1-xy)^{-n}(1-y)^{-1} - (1-yx)^{-n}(1-y)^{-1} + (1-yx)^{-n}$

V $(1-x)^{-1}(1-yx)^{-n}(1-x)^{-1} - (1-x)^{-1}(1-xy)^{-n}(1-x)^{-1} - (1-yx)^{-n}(1-x)^{-1} + (1-x)^{-1}(1-xy)^{-n}$

VI $(1-x)^{-1}(1-yx)^{-n}(1-y)^{-1} - (1-x)^{-1}(1-xy)^{-n}(1-y)^{-1} - (1-yx)^{-n}(1-y)^{-1} - (1-x)^{-1}(1-yx)^{-n} + (1-x)^{-1}(1-xy)^{-(n-1)}(1-y)^{-1} + (1-yx)^{-n}$

VII $(1-y)^{-1}(1-yx)^{-n}(1-x)^{-1} - (1-y)^{-1}\{S + 1\}(1-x)^{-1} + S(1-x)^{-1} + (1-y)^{-1}S - S$
where $S = (1-xy)^{-n} + \ldots + (1-xy)^{-1}$

VIII $(1-y)^{-1}(1-yx)^{-n}(1-y)^{-1} - (1-y)^{-1}(1-xy)^{-n}(1-y)^{-1} + (1-xy)^{-n}(1-y)^{-1} - (1-y)^{-1}(1-yx)^{-n}$

⌘

[6] Each element, g, is replaced by N(g,G-{g}) -- and omitted if it reduces to 0.

[7] The observant reader will only count 10 stars. Two of the starting relations reduce to 0 by other elements of the Gröbner Basis, and so are omitted.

DEDUCTION:

The construction of Gröbner Bases, starting with a set of identities, provides an example of deduction. The set of starting identities, G, are known to be true (they usually are just obvious statements about inverses). Any new element in the Gröbner Basis G' is therefore also known to be an identity (since it is a linear combination of identities). Also any f for which N(f,G') = 0 is an identity. A formal proof of the new identity can be obtained by tracing the basis computation to provide the linear combination explicitly.

EXAMPLE 5

The resolvent identity, $(1-x)^{-1}x^{-1} - (1-x)^{-1} - x^{-1}$ is a classical identity in operator theory. It is not difficult to prove. However it was produced automatically (with an implicit proof) because it occurs in the Gröbner Basis for RESOL (see previous example). Therefore it must be a linear combination of these identities. In fact $(1-x)^{-1}x^{-1} - (1-x)^{-1} - x^{-1} = x\, g_2 - g_3\, x^{-1}$

We have just examined the case in which the starting relations are identities. In this case, the Gröbner process produces more identities and also provides an algorithmic test for a polynomial expression to be a linear combination of a set of identities, and hence an identity itself.

Some theorems in operator theory can be stated in such a way that the hypotheses and conclusion involve the vanishing of polynomial relations (see [HSW2] for examples). In this case, we are trying to deduce that f=0, but not universally -- only under the condition that the underlying matrices satisfy $\{g_i = 0\}$. Any linear combination of the g_i vanishes where the g_i vanish, so the members of the Gröbner Basis, G', are consequences of the hypotheses. If N(f,G') = 0 then f =0 follows from $\{g_i = 0\}$.

EXAMPLE 6 $f = [y(1-xy)^{-1} - (1-yx)^{-1} + 1][(1-xy)^{-1} - 1]$

f is a polynomial in the 4 variables x, y, $(1-xy)^{-1}$ and $(1-yx)^{-1}$ It is not an identity. We can show, however, that f = 0 provided x and y are idempotent (i.e. $x^2 = x$, $y^2 = y$).

Let $G = \{ (1-xy)(1-xy)^{-1} - 1, (1-xy)^{-1}(1-xy) - 1, (1-yx)(1-yx)^{-1} - 1,$
$(1-yx)^{-1}(1-yx) - 1, x^2-x, y^2-y \}$

The Gröbner Basis, G', for G has 18 elements. The division process shows N(f,G') = 0. Thus f is a linear combination of the polynomials in G and so f=0 for any x, y satisfying them.

Part III - Forth Based Systems for Math Research

Forth is useful as a base language for user-created mathematical research systems. It provides the tools (e.g. low level access to hardware and language) and the features (e.g. simplicity, control over compilation, high level constructs) for building high level languages and interactive systems. Research systems can be constructed which are tailored to the needs of a specific project or area. They can be used interactively, are easily extended, and are easily modified. Flexibility is achieved without great sacrifice in execution speed. In this part we discuss some aspects of the Forth-based system used in research on Gröbner Basis Theory.

DATA ABSTRACTION AND MODULARITY:

Algebra programming often requires the ability to simultaneously handle a great many types of data. The system used for this work handles integers, addresses, high-precision integers, high-precision rationals, strings, ordered lists, pointers, and polynomials. It was decided that all objects in a system should be manipulated on the stack as single stack cells -- and subject to the ordinary stack manipulations of Forth. (Thus `A B SWAP` should interchange `A` and `B` no matter what type of objects these are.)

The most prominent objects in our previous discussion are polynomials in several variables. If the variables commute, we can write a polynomial (say of three variables) in the form:

$$f = \sum a_{ijk}\ x^i y^j z^k$$

These are added and multiplied using the familiar rules of polynomial arithmetic. The a_{ijk} are coefficients which lie in some specific domain which has addition and multiplication (integers, rationals, integers mod p, etc.). If the variables do not commute, then polynomials look like

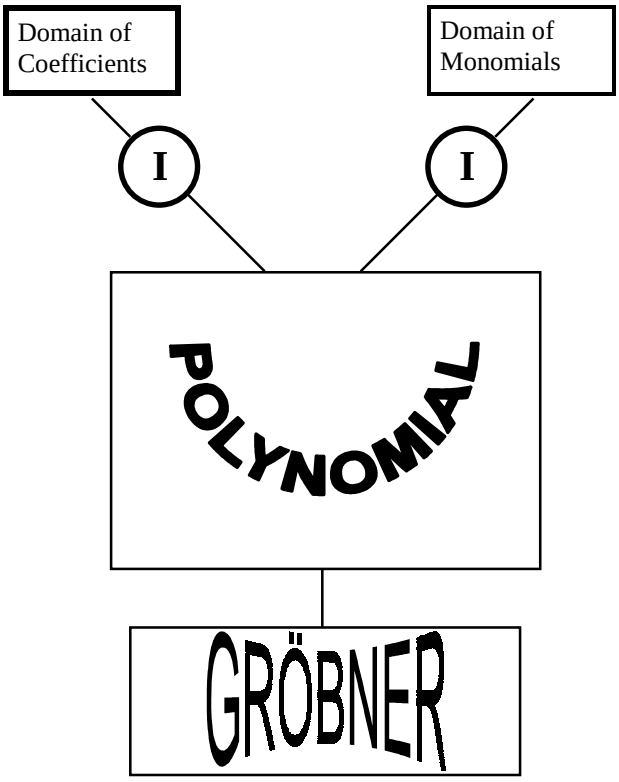
$$f = \sum a_i\ W_i$$

Where the W_i are "words" in the variables. (Example: f = xxy - 2xyx + 3 yxx)

When polynomials are multiplied, the words are concatenated in the non-commutative case -- while the exponents are added in the commutative case. Since polynomials of various types occur so frequently in algebra, it is useful to implement an abstract data type "Polynomial" as a module which can be linked to modules implementing the coefficients and the "monomial type". This module, in turn, is linked to client modules which use polynomials (like a module to implement the Gröbner algorithm).

A polynomial is a finite sum of terms. Each term always consists of a coefficient and a monomial part. The arithmetic of polynomials depends on the arithmetic of the coefficient domain and a "multiplication" of monomials. An interface module provides the

polynomial module with a standard description of the coefficients: the coefficient arithmetic operations are given by operations `X+`, `X-`, `X*`, and `X/` which always take two coefficient objects from the stack, and return a coefficient object. Similarly there is a `MONO*` which takes two monomial objects from the stack and returns a monomial object. The arithmetic of polynomials can be described abstractly in terms of these operations together with a few others.



A polynomial itself will be a circular linked list whose nodes contain, or point to, a coefficient object and a monomial object. Polynomial operations are implemented by running pointers along these lists. An abstract polynomial module will manage the storage for this representation and provide for polynomial arithmetic, input, and output. To do this, it is written in terms of operations which it imports from the modules which implement the coefficient and monomial domains. It, in turn, exports polynomial arithmetic operations (like `P+`, `P-`, `P*`, `P.`) as well as access words (like `COEFF@`, `MONO@`, `COEFF!`, `MONO!`).

The same package can be used to produce a variety of different polynomial types. We just first load the module implementing the coefficients and monomials followed by a small module interface module (indicated by **I** in the diagram) that tells the polynomial module what it needs to know about coefficients. Similarly, an interface module provides the polynomial package with knowledge of monomials. In the system used for the current work, an order is imposed on the monomials and the terms are kept in descending order, so the polynomial package must need to know how to compare two monomials.

Extended precision integer and rational arithmetic is used heavily in algebraic computation. For most investigations, the coefficients were extended precision rationals implemented in a package I created over 9 years ago. An extended precision rational package requires a memory management scheme to be useful. I used a method which provides a collection of 16 storage locations for temporaries used in a conjunction with a program to manage them. The same storage management package is used for extended precision integers, extended precision rationals, strings, and polynomials. [Wav]. The basic arithmetic words are written in assembly language.

Forth encourages a writing style in which functionality is distributed over a relatively large number of specialized "words". It therefore encourages (although does not coerce) data abstraction: the hiding of details about implementation of data structures from client words which use the data structures. The top level of this system appears to manipulate mathematical objects on the stack. Lower level words implement the data structures and export the "access words" for manipulating them. Details of implementation are encapsulated and so can be changed without affecting other parts of the code.

A number of changes have taken place in the implementation of the polynomial package since it was first written in 1984. The value of hiding implementation details can be illustrated by mentioning the changes that have taken place in just one area: representation of terms and memory allocation:

In 1984 the package was intended to perform simple computations on polynomials in non-commuting variables. This imitated an earlier package for polynomials in commuting variables. Everything was in the 64k code segment. The terms of a polynomial were represented by nodes of linked lists which consisted of a link field, coefficient field (integer), and monomial field (string of length up to 32). A pointer was a variable containing the address of a node. The access word `COEFF@` was defined by

```
: COEFF@  ( ptr -- coeff)  IA + ;
```
[8]

The commutative case of the Gröbner Basis algorithm was implemented in 1986 using the abstract polynomial idea. Capacity was increased by moving data to separate segments outside the code segment. The nodes, for example, were in a separate 64k "pointer segment" and pointers were variables containing offsets in that segment. To hide information about where the data for pointers is located, all of the words involving them use two basic access words `PT@, PT!` which are used just like `@` and `!` (in fact, are `@` and `!` if pointers lie in the code segment). Putting a class of objects into its own segment and addressing objects by offset allows objects to be represented on the stack by a single cell. This approach seems preferable, for portability and ease of programming, to using a two-

⌘———————————————

[8] IA is the number of addressable units needed to represent an integer. It is what we now call CELL. This notation was inherited from Kitt Peak VAX-Forth.

cell (segment-offset) representation. Code is clearer if objects have their own access words -- rather than using general-purpose architecture dependent words.

The nodes, which lie in the same segment as the pointers, consisted of a link field and the segment/offset pair of the coefficient and monomial. The access word COEFF@ became:

```
: COEFF@ ( ptr -- x )    \ transfer coeff from node to temp
      DT@  ( offs seg ) \ location of coeff
      XTEMP            \ address of temporary
      DUP >R  XSIZE  F>MOVE  R> ;
```

XTEMP returns the address of the next available storage location for coefficients. A coefficient is moved to a temporary location in the code segment, and the address is returned.

When writing words for dealing with polynomials as linked lists, it is important to create a set of words which have conceptual appeal. For example, we often advance a pointer along the terms of the polynomial (nodes in the list). Thus we define

```
: ADVANCE ( ptr -- ) \ set pointer to next term
        DUP PT@ PT@ SWAP PT!  ;
```

Where we assume that the pointer points to the link field of a node. Psychologically, this is a word which makes the following change:

The non-commutative Gröbner algorithm was implemented in 1990. Near the end of the year, work started on the operator theory project. Eventually more memory was needed, and the system was modified to use “expanded memory” which involves paging. Nodes now contain the offset and page of the data (coefficient and monomial). COEFF@ looks the same, but DT@ now must get the offset/page, map the page to a buffer, and return the offset/segment of the data within the page buffer.

It may be argued that most of the problems above were inflicted by the idiosyncracies of the Intel segmented architecture -- but what is important is that Forth supports a writing style which conceals idiosyncracies of hardware within the data access words. If a system is carefully written, there will be less that a dozen words that really need to know where the coefficients and monomials are actually located in memory or how memory is configured. This information is contained in words like COEFF@ and COEFF!. Thus the changes mentioned above were made with almost no impact on the rest of the code.

IMPLEMENTING INFINITE BASES:

We previously gave an example in which the Gröbner Basis is infinite. In this example there are 8 infinite parametrized families of polynomials together with a relatively small set of special polynomials that do not fall into these classes. The Division algorithm is easily extended to the case in which G is infinite provided that the decision Divides(LT(g),t) can be made in finite time. In an allowable term ordering, if M_1 divides M_2, then $M_1 \leq M_2$. In the ordering we use, $M_1 \leq M_2$ implies length(M_1) ≤ length(M_2). Thus, for a given monomial, there are only a finite number of candidates for divisors. In a reduced Gröbner Basis, the LT(g) are distinct. So there are only finitely many candidates for g such that Divides(LT(g),t).

The preceding paragraph shows that to reduce polynomials whose terms have length bounded by N, it suffices to have stored in memory all basis polynomials with leading terms ≤ N. Thus only a finite number of basis elements are needed to reduce polynomials below a given order. In practice, the fact that the basis is infinite can be ignored.

For experiments supporting theoretical work, however, it became necessary to apply the division algorithm for polynomials of arbitrarily high order. It was not feasible to generate and store all the basis polynomials that would be needed. A method was devised to perform reductions using an infinite basis.[9]

It is simple to implement parametrized families like those above: we define a word, GR (n m -- poly) which generates the n^{th} element of the m^{th} family upon demand. If the polynomial G(n,m) is not in memory initially, it can be created. Notice that the length of LT(g) increases with n -- so to test Divides(LT(g),t) only a finite number of these polynomials would need to be generated. This shows that carrying out division by our Gröbner Basis G is at least a task that can be carried out in finite time.

Now we can become concerned with efficiency. We wish to avoid generating any GR(n,m) which is not needed. Remember that only the leading term is used to check divisibility -- so the test for divisibility does not require generation of the polynomial. The polynomial is only needed when a multiple of it is to be subtracted. Here is how the divisibility decision, based on leading terms, is implemented.

Internally, the polynomials variables are denoted a,b,c,... Thus, in our example, a = x, b = y, c = x^{-1}, d = y^{-1}, e = $(1-x)^{-1}$, f = $(1-y)^{-1}$, g = $(1-xy)^{-1}$, h = $(1-yx)^{-1}$. The leading terms are:

⌘———————————

[9] It should be noted that this method is applied ad hoc, using information about the particular basis. The process of identifying families, finding formulas, and implementing reduction for an infinite bases has not yet been automated.

General Classes		Special Relations					
I	ah(n)e	1	ac	9	df	17	hb
II	ah(n)f	2	ae	10	dh	18	hc
III	bg(n)e	3	bd	11	ea	19	abg
IV	bg(n)f	4	bf	12	ec	20	bah
V	eh(n)e	5	ca	13	fb	21	ebg
VI	eh(n)f	6	ce	14	fd	22	fah
VII	fh(n)e	7	cg	15	ga		
VIII	fh(n)f	8	db	16	gd		

To check for divisibility a monomial was scanned for occurrence of pairs of letters which start these monomials. What happens next depends on the pair. If ah, for example, the number of successive h was counted and then the routine looks for a terminating e or f. A hash coding scheme for the pairs was used to quickly classify the pair and determine further treatment. The result of this scan is the precise first location of LT(g) as a substring in t, and the precise g, or else a FALSE flag.

FLEXIBILITY:

The most important attribute of a Forth-based system is flexibility. The ability to easily modify and extend the language is extremely important for research systems. It is rarely clear, at the start of a project, what will be discovered and the tools which will be needed to discover it. The software system must therefore adapt to the project's needs. The system used for this project, in addition to changes in data representation, acquired several sets of input and output words as it was used. My colleague and his students are incorporating the results of this work in a Mathematica package for engineers, thus a provision was added to accept input and provide output in the form of Mathematica expressions. When we reached the stage of writing papers, I added the ability to produce output in TeX (to minimize transcription errors in final papers). One of the major problems in this project was the interpretation of output. Special output words were implemented to assist in this task. I also needed to trace and experiment with some steps in the algorithm -- this was easily accomplished by changing the behavior of key words. It was important to me that a Forth system allows modifications to be made incrementally, loading supplementary code (or even patching existing code) without recompiling the entire system and losing the existing data.

SPEED:

While execution speed is not the most important advantage Forth has to offer, it should be noted that Forth-based systems are substantially faster than commercial computer algebra systems (like Mathematica, Maple, Macsyma, REDUCE, and Derive). Some timing studies showed that the Forth-based system (on a 486/33) performs arithmetic on polynomials in non-commuting variables over 20x faster than my colleague's Mathematica package (running on a SPARC II). Some of the basis computations which

took several hours on the Forth-based system would have taken several days using Mathematica.

CONCLUSION:

Algebraic computation benefits from a dynamic view of language and system. The needs of someone exploring new algorithms and new applications are best met by providing the user with tools for creating and modifying both the language and the system. Forth provides a good foundation for this model of computation.

References

[Buch1] B. Buchberger, "Ein Algorithmus zum auffinden der basiselemente des restklassenringes nach einem nulldimensionalen polynomideal", *Doctoral Dissertation Math Inst University of Innsbruck, Austria*.

[Buch2] B. Buchberger, "Gröbner bases: an algorithmic method in polynomial ideal theory", *Recent Trends in Multidimensional System Theory*, Reidel (1985), pp. 184-232.

[Hiro] H. Hironaka, "Resolution of singularities of an algebraic variety over a field of characteristic zero: I, II", *Annals of Math*, **79** (1964), pp 109-326.

[HW] J. W. Helton and J. J. Wavrik "Rules for computer simplification of the formulas in operator model theory and linear systems", *Operator Theory: Advances and Applications*, **73** (1994), pp. 325-254.

[HSW1] J. W. Helton, M. Stankus and J. J. Wavrik "Computer simplification of engineering systems formulas", *Proc IEEE Conference on Decision and Control* (1994), pp. 1893-1898.

[HSW2] J. W. Helton, M. Stankus and J. J. Wavrik "Computer simplification of engineering formulas", submitted to *IEEE Transactions on Automatic Control* (preprint 35 pages, 1995).

[Mora] F. Mora, "Groebner Bases for non-commutative polynomial rings" *Lecture Notes in Computer Science*, **229** (1986), pp. 353-362.

[Wav] J. J. Wavrik, "Handling multiple data types in Forth", *Journal of Forth Application and Research*, **v. 6 no 1** (1990), pp 65-76.