IFIP

International Federation for Information Processing

Working Conference on Programming Environments for High–Level Scientific Problem Solving

23 - 27 September 1991, Karlsruhe, Germany

23/09/91 Introduction

Page	Speaker
1	Siu Shing Tong
18	W. Morven Gentleman
20	Paul C. Abbott

23/09/91 Intelligent Assistants

Page	Speaker
30	Siegfried M. Rump
35	Willi Schönauer
46	John A. Nelder
51	Ronald F. Boisvert

24/09/91 Mathematical Methods

Page	Speaker
61	Masaaki Shimasaki
71	Vladimir P. Gerdt
80	Bruno Buchberger
92	Dominique Duval

24/09/91 Systems and Tools

Page	Speaker
101	Stuart I. Feldman
106	Hans Zima
116	Wayne R. Dyksen
125	Neil Hurley

25/09/91 Systems Engineering I

Page	Speaker
137	Michael Clarkson
139	Alfonso Miola
149	Elias Houstis
164	James M. Purtilo

26/09/91 Interaction & Visualization

Page	Speaker
174	Yukio Umetani
183	Eric Grosse
189	Hans J.W. Spoelder
196	Peter M. Dew
212	Colin W. Cryer
222	Steve J. Hague
242	Niklas Holsti
246	Fred Grossman
252	David Y. Y. Yun
261	J. A. van Hulzen
270	Yurij I. Shokin
274	Allan Bonadio

27/09/91 System Engineering II

Page	Speaker
279	Kevin A. Broughan
280	I. A. Nicolayev

Prepared by Bergen Scientific Centre IBM

Gröbner Bases in Mathematica: Enthusiasm and Frustration.

Bruno Buchberger
RISC-LINZ

—

Research Institute for Symbolic Computation
Johannes Kepler University, A4040 Linz, Austria (Europe)
Tel: +43 (7236) 3231-41, Fax: +43 (7236) 3338-30
Bitnet: k313370@AEARN

—

February 8, 1991

Abstract

We report on an experimental implementation of Gröbner bases in Mathematica. This experiment gives insight into the performance of Mathematica as a scientific system for algorithm researchers. We draw two major conclusions:

- We are enthusiastic about Mathematica with respect to the programming style it supports, which allows easy, well-structured, and generic implementation of algorithmic ideas.
- We are frustrated because Mathematica is so extremely slow that its use for scientific experiments of serious size is prohibitive.

1 The Experiment and Its Result

The experiment described in this paper is part of the book project (Buchberger 1991). The book on which the author is working intends to provide an easy, but mathematically complete, introduction to the theory of Gröbner bases and its many applications. We also intend to distribute, along with the book, a GRÖBNER software package in source code. The goal of GRÖBNER is threefold:

- *students* should be supported in learning the theory by having the possibility to try out examples and to see all details of an implementation,

- *researchers* should be able to experiment with new versions of the algorithms, check hypotheses, expand and improve the package,
- and *users* should be encouraged to apply the method for various concrete large-scale problems.

A suitable language for GRÖBNER should therefore meet the following requirements:

- the language should be *available on as many machines as possible*,
- the language should be *professionally distributed and supported*,
- the code should be *easily readable*,
- some *basic algebraic algorithms* (long integer arithmetic and, in refined versions of the package, rational function arithmetic and polynomial factorization) should already be available in the language,
- the language should support *generic programming* (formulation of algorithms independent of the underlying data domain),
- the *code should be fast.*

I compared systematically a number of available languages that might be considered as an immediate choice: C, C^{++}, LISP, PCL (Portable Common Loops), muSimp, SAC-2, Scratchpad-2, Maple and Mathematica. After extensive experiments with coding all or part of GRÖBNER in these languages, I found that the appropriateness of these languages for the task at hand can be summarized in the following table:

	C	C^{++}	Lisp	PCL	muSIMP	SAC-2	Scratch-pad-2	Maple	Mathematica
availability	+	+	+	∓	-	+	-	+	+
professional distribution	+	+	+	∓	+	-	+	+	+
readability	±	±	±	±	+	+	+	+	+
algebraic algorithms	-	-	-	-	±	+	+	+	+
genericity	-	+	-	+	-	-	+	-	+
speed	+	+	+	∓	+	+	∓	-	−

(Maple will soon have generic programming facilities.)

The coarse scale used in this table is: $+, \pm, \mp, -$. The last line concerning speed can be given in more detail:

	C	C^{++}	Lisp	PCL	muSIMP	SAC-2C	Spad-2	Maple	Mathematica
speed	1	1/2	1/10	1/100	1/10	1	1/100	1/1000	1/3000

This line should be understood in the way that, for example, a program written in Mathematica as a language (not a call to an eventually available built-in function) is approximately 3000 (three-thousand!) times as slow as the same program written in C. One main part of this paper (Section 3) will be devoted to backing this assertion.

When I started my experiments with the above candidate languages for GRÖBNER, I already knew most of the entries in the above rough table of performance criteria because most of this is well know and documented. The two entries that surprised me most (and it took me quite some time to "fill these entries in" because very little is said about this in the official documents and critical assessments) were

- the strength of Mathematica for generic programming and
- its prohibitively slow speed.

Therefore, I would like to devote this paper to a discussion of these two aspects.

The conclusions we draw are, of course, independent of the particular package considered. Only for the examples we will need some basic knowledge about Gröbner bases, see the survey article (Buchberger 1985). It is clear that the present paper cannot give an introduction to Mathematica either. For all details about Mathematica see the document (Wolfram 1988).

2 Programming Style and Generic Programming

The programming style of Mathematica is elegant mainly because of two reasons:

- Mathematica's fundamental data type is the "expression", which models both nested data and nested function descriptions and encompasses the expression encounterd in ordinary mathematics in a very natural way.
- In function definitions, Mathematica allows arguments that are "patterns" (i.e. terms) and not only variables. Thus, one often can formulate algorithms in Mathematica without explicit "selectors" and "constructors", which tend to make programs clumsy in other languages. Again, this pattern matching programming style is what normally is used in ordinary mathematics (or in its formalization in predicate logic) for describing algorithms.

For example, any selection of the ordinary mathematical rules for "lim" would immediately yield an executable Mathematica program:

```
Limes[a_+ b_] := Limes[a] + Limes[b]
Limes[a_ b_] := Limes[a]  Limes[b]
```

Every Mathematica expression that matches the "pattern" `Limes[ a_+ b_]`, where `a_` and `b_` stand for arbitrary expressions, would be transformed by the Mathematica interpreter according to the first rule. In fact, `Limes[ a_+ b_]` is an abbreviation for `Limes[ Plus[ a_, b_]]`. Even, an entire rule like `Limes[a_+ b_] := Limes[a] + Limes[b]` is in fact a single expression, which in "full form" would be

```
Define[ Limes[ Plus[ Blank[ a], Blank[b]]],
    Plus[ Limes[ a], Limes[ b]]]
```

Generally speaking, except for some atoms, the only data items in Mathematica are expressions of the form $f[e_1, \ldots, e_n]$, where $f, e_1, \ldots, e_n$ are again expressions.

This simple concept of "expression and matching" is very powerful. We will now show that it incorporates, in a very natural way, the concept of "generic programming", which is vital for complex algebraic packages. We find it worthwhile to expand on this point because, although briefly mentioned in the document (Wolfram 1988), it is not so commonly known.

The main point how generic programming can be incorporated in Mathematica programs is the fact that the f in a Mathematica expression $f[e_1, \ldots, e_n]$ can be used as a "tag" for characterizing a data domain. Objects having two different (constant) tags `T1` and `T2` will automatically be analyzed to belong to two different domains and, accordingly, two different sets of rules may be installed for the same operation `Operation`:

```
Operation[ T1[ ...]]   := first right-hand side,
Operation[ T2[ ...]]   := second right-hand side.
```

When finding an expression of the form `Operation[` *expr*`]`, the Mathematica interpreter analyzes the expression *expr* and depending on whether *expr* starts with `T1` or `T2` applies the first rule or the second. This simple mechanism can be used to create "generic packages" that handle complicated "towers" of algebraic domains without repetition of code.

For example, when implementing Gröbner bases (over multivariate polynomials) one would like to formulate the algorithms for a wide range of different domains of coefficients and for many different orderings and various representations of power products. Therefore it is not possible to use the built-in multivariate polynomial package of Mathematica (which is actually coded in C and is not available in source code for users!)

One way of organizing the many possible "towers" of domains for GRÖBNER is as follows: The top-most domain is the domain of distributive polynomials which I chose to represent in the following "nested" form:

DNP[*m*, *dnp*],
where m is a monomial and
dnp is again a distributive polynomial.
(DNP[ ] is the "empty" (zero) polynomial.)

The monomials might be represented in the following form

Mon[*c*, *pp*],
where c is an element of a coefficient domain and
pp is an element of a power product domain.

We are interested in many different coefficient domains: the built-in rational numbers, the finite fields GF(p), the field of rational functions etc. For example, finite field elements in GF(p) might be represented by

FF[*f*],

where f is a number modulo a prime p, and rational functions may be represented by

RF[*rf*],

where *rf* might be a rational function in the built-in Mathematica representation.

Finally, the power products may be represented as "exponent lists" in the form

EL[$e_1, \ldots, e_n$],

where the $e_1, \ldots, e_n$ are the exponents at the n indeterminates. Alternatively, one may be interested in a "Gödel coding" of the exponents $e_1, \ldots, e_n$ by the natural number $p_1^{e_1} \cdots p_n^{e_n}$, where the $p_1, \ldots, p_n, \ldots$ are the prime numbers. A corresponding representation in Mathematica may be

GE[*ge*],

where *ge* is a natural number.

Now we show some parts of the corresponding Mathematica code for realizing arithmetic on these domains:

Addition for DNP*-polynomials*:

```
dnp_DNP + DNP[] := dnp

DNP[ m1_, dnp1_] + DNP[ m2_, dnp2_] :=
    DNP[ m1, dnp1 + DNP[ m2, dnp2]] /; m1 > m2

...

DNP[ m1_, dnp1_] + DNP[ m2_, dnp2_] :=
    DNP[ m1 + m2, dnp1 + dnp2] /; IsEquivalent[ m1, m2] && m1 != -m2
    ...
```

(A rule containing a variable like dnp_DNP on the left-hand side must be read as follows: If an argument dnp whose tag is DNP is encountered then the rule is applied). Note that the "+" on the left-hand side of these rule denotes the addition specific for DNP-polynomials whereas the "+" on the right-hand side is "generic" in the sense that, at run-time, the objects m1, m2, dnp1, dnp2 etc. are analyzed and, in dependence on their "tag" (in this case "Mon" or "DNP"), the appropriate rule of the package is selected and applied. Similarly, in this example, also ">", "−", and "IsEquivalent" are "generic".

Some operations on Mon*-monomials*:

```
Mon[ c1_, pp1_] > Mon[ c2_, pp2_] := pp1 > pp2
Mon[ c1_, pp_] + Mon[ c2_, pp_] := Mon[ c1 + c2, pp]
IsEquivalent[ Mon[ c1_, pp1_], Mon[ c2_, pp_]] := pp1 == pp2
```

The operations ">", "+", and "IsEquvialent" appearing on the left-hand side of these definitions are the specific realizations for Mon-polynomials of the generic operations used in the above domain of DNP-polynomials. The operations ">" and "+" on the right-hand side are again "generic".

Let us consider one more "layer" in this example of a "tower of algebraic domains".

">" on EL*-power product*:

```
el1_EL > el2_EL :=
    Block[ i, ...,
        For[ i = 1, ...
            ...
            If[ el1[[ i]] > el2[[ i]],
                ...
                ]
            ]
        ]
```

">" on GE*-power products*:

```
GE[ e1_> GE[ e2_] := e1 > e2
```

In these two definitions, again, ">" on the left-hand side denotes the operation specific for the domain of EL-power products and GE-power products, respectively. ">" on the right-hand side denotes the corresponding generic operation. Typically, the domain of natural numbers will be used as the substitute for these generic domains. However, it is well conceivable that other domains are used as exponent domains. For example, it may turn out that "symbolic exponents" (i.e. polynomial or rational expressions) are an interesting exponent domain.

Addition on rational number coefficients: Arithmetic on these numbers is built-in. In fact, these numbers have the internal tag Rational.

Addition on finite field elements:

```
FF[ f1_] + FF[f2_] := FF[ Modul[ f1 + f2, $Prime]]
```

Addition on built-in rational functions:

```
RF[ rf1_] + RF[ rf2_] := RF[ Factor[ rf1 + rf2]]
```

In the last two examples, again, "+" on the left-hand side is specific for the domains "FF" and "RF", respectively, whereas "+" on the right-hand side is generic. Typically, in these cases, the generic rule will only be applied to the domain of integers and the domain of built-in rational function expressions, respectively.

On top of the DNP-polynomials, a number of higher levels in the generic domain hierarchy are constructed in GRÖBNER, for example, the domain of sets of polynomials, pairs of polynomials, sets of pairs of polynomials etc.

A package that is constructed according to the above principle can then be used for a huge variety of domain combinations. At run-time, the Mathematica pattern matcher will analyze the tags of the data expressions and select the appropriate rules in the rule base (= "program"). Roughly, if in a package with m "layers" of domains there are n domains in each layer then the package can be used for n^m many concrete domains although there are only $m \cdot n$ many pieces of code!

Summarizing, I think that generic programming in Mathematica along the above lines for towers of algebraic domains is elegant, natural, versatile, and yields intelligible, easy-to-change and short code. I really enjoyed programming in this style.

In addition, the slow-down caused by tagging objects is tolerable. By appropriate distribution of the code (how this can be controlled by the programmer is described in the Mathematica document), as a rule of thumb the slow-down is approximately by the factor of 2 if one has 10 rules in each domain.

Having adopted the above generic style for a preliminary version of GRÖBNER I was nearly convinced to stay with Mathematica for carrying the project through. I drastically changed my mind when I started systematic measurements of computing time: It turned out that, whereas the relative slow-down by generic programming is tolerable, the absolute computing times are prohibitive. I report on this in the next section.

3 Speed

Speed is important both for the *user* of a system like GRÖBNER and for the *researcher*.

Algebraic methods like Gröbner bases, which are "universal" for a broad class of problems, tend to be "exponential" in their behavior. High speed is therefore essential for *using* the method in practical cases. For example, good results have been achieved recently in robot kinematics by using the Gröbner bases method on the PSI machine at ICOT (Tokyo), see (Sato, Aiba 1991). The computing times are in the range of several seconds, which is quite tolerable for this kind of problems. A slow-down by a factor of 1000 or even "only" 100 by using the wrong language would make the application worthless.

However, also the *researcher* working in such an area heavily depends on speed because he needs to study huge series of test examples for observing the dependence of computing time on input parameters, for studying certain phenomena in the intermediate results that may lead to new conjectures and eventually to new theorems, and also for using the method as a building block for other algorithmic problems. For example, recently the Gröbner bases method is heavily studied as a building block in the context of Zeilberger's approach to the automated generation and proof of combinatorial identities and and the computation of definite sums and integrals, see (Takayama 1990).

For avoiding misunderstandings, let me point out that Mathematica (and similar systems like MAPLE) may be very fast if one uses the *built-in* C functions. Externally, these functions can be called by Mathematica function calls. Internally, however, they are not written in the Mathematica language but in C. Their code is not accessible and even if it would be accessible it would be of little use for the tutorial and research purposes described in this paper.

Even this favorable statement about the speed of built-in functions in Mathematica must be relativized because some of the functions, in particular the built-in Gröbner bases function, perform fast on small examples but show an unexplained increase in computing time on slightly bigger examples. This is absolutely intolerable for using the system as a research tool. Not only does the unexplained increase in computing time lead to the conclusion that the implementation does not use all theory available for the method but it leaves the researcher with absolutely no possibility to analyze the reason for the unexplained behavior. Also, it is not possible to adjust the built-in functions to changing needs, for example in the case of GRÖBNER, to variable coefficient domains and variable admissible orderings of power products.

In a situation like ours where we want to distribute software in source code for students, researchers and users in fully documented form, the speed of Mathematica must be judged by its performance for algorithms that are fully formulated in Mathematica with any resort to undocumented algorithms written in C. Since my initial experiments with the speed of parts of GRÖBNER written in Mathematica were so disappointing I went through a detailed time analysis of the fundamental Mathematica operations. Summarizing what I found is:

Class of Operations	Time per operation in millisec (on an Apollo 3500)
constant time operations on Mathematica "lists" (i.e. arrays internally): e.g. Length, First, Last, Part	1.5
constant time operations on "nested lists": e.g. FirstN, RestN, PrependN	3
Iteration over Mathematica "lists": e.g. Rest, Drop, Prepend, Append, Insert, Reverse	1 + 0.04 l
Map and Scan iteration over "lists"	1.5 l
user programmed iteration over "lists" using "For" etc.	3 l
user programmed iteration over "nested lists" LengthN, ReverseN, MapN, ScanN etc.	6 l

The parameter "l" in the above table denotes the length of lists. In order to understand the length-dependent complexity of some of the above operations one must know that Mathematica "lists" internally are represented by arrays. (This is nowhere documented in the Mathematica publications. However, for algorithm researchers this is very important information.)

The "nested lists" mentioned in the above table are expressions of the following kind:

```
T[ e1, T[ e2, T[ e3, ...  T[ ] ...]]],
```

where T is some "tag" and the ei are the actual elements of the list. This may be used as one possible simulation of true "lists" in Mathematica. The user-defined operations on such nested lists have the suffix "N" in the above table.

From the above table one sees that the Mathematica operations are approximately 3000 - 8000 times slower than the corresponding operations programmed in C. (The Apollo 3500 is a 5 MIPS machine). (My experimental results are also backed implicitly by the examples in (Maeder 1990), which partly contain timings. However, no explicit mention about speed is made in the official Mathematica documents!)

As a consequence, high-level algorithms fully written in the Mathematica language and not resorting to built-in medium grain algorithms like polynomial arithmetic are prohibitively slow. Here are the computation times of two typical Gröbner bases examples:

	3 variables 3 polynomials degree 2	3 variables 3 polynomials degree 4
My implementation on a ZUSE Z23 (1965!) in assembler code	2 min	12 min
Built-in Mathematica Gröbner basis function on Apollo 3500	1 sec	> 2 days
Built-in Maple Gröbner basis function on Apollo	4 sec	2 min
My implementation fully coded in Mathematica language	5 min	30 min

From this table one sees that the speed-up of approximately 1000 achieved by hardware improvements in the last 25 years (the ZUSE Z23 was a 0,003 "MIPS" machine!) is completely lost by the software elegance of Mathematica and similar high-level languages.

Also, one sees that the undocumented built-in Gröbner bases function of Mathematica performs very well on small examples but shows an unexplained increase in performance for slightly larger examples. (This phenomenon was observed by many researchers with some other of the built-in Mathematica functions).

The reasons for this bad speed performance of Mathematica as a language most probably are manifold:

- Mathematica is interpreted, not compiled,
- a function call in Mathematica, since it performs something so general as pattern matching, needs approximately 1 millisecond independent of the function body,
- in integer arithmetic the most general case is assumed even if only index computations are executed,
- others?

I think that these reasons should be analyzed in more detail and documented in future Mathematica documents.

4 Conclusions

In this paper I concentrated on only two criteria for judging Mathematica. Many other criteria could be considered, see the extensive critical analysis (Fateman 1990). Interestingly enough, in (Fateman 1990) the two criteria, on which I concentrate in this paper and which are the crucial ones for judging a system as an instrument for algorithm research, are not treated in any detail.

From what I learned in the above experiments I drew the following conclusions:

- Although there are many interesting software systems available for computer algebra, the "ideal" system does not yet exist. The ideal system would combine the elegance and naturalness of Mathematic and the speed of C. I know this is impossible but I think we could achieve something much better than what exists now. For example, it would already help a lot if there was a possibility to incorporate user-defined C routines in high-level languages like Mathematica.
- For my own GRÖBNER project I now decided to use Collins' SAC-2 system in a new version that is entirely coded in C. I designed a simple preprocessing mechanism that allows a rudimentary form of generic programming that seems to be sufficient for a project like GRÖBNER. I will report on this in a subsequent paper. G. Collins and I decided to cooperate on turning "SAC-2C" into a professionally distributed system such that many researchers can use it in situations like GRÖBNER where algorithm researchers want to distribute easy-to-read, generic and fast source code.

Acknowledgment: This paper was written in the frame of the "Gröbner Bases Project" sponsored by the Austrian Ministry of Science and Research. I would also like to thank Roman E. Maeder for pointing out the possibility of generic programming in Mathematica to me. Thanks also to Stephen Wolfram who provided a test installation of Mathematica for the "Gröbner Bases Project".

References

B. Buchberger. Gröbner bases: an algorithmic method in polynomial ideal theory. In: *Multidimensional Systems Theory* (N. K. Bose editor), pp. 184 - 232. D. Reidel Publishing Company, Dordrecht - Boston - Lancaster, 1985.

B. Buchberger. *Gröbner Bases: Elementary Theory and Applications.* Springer-Verlag, New York - Vienna, in preparation.

R. Fateman. *A Review of Mathematica.* Manuscript. Dept. of Computer Science, University of California, Berkeley, 1990.

R. E. Maeder. *Programming in Mathematica.* Addison - Wesley Publishing Company, Redwood City, California, 1988.

S. Sato, A. Aiba. *An Application of CAL to Robotics.* Manuscript, ICOT Institute, Tokyo, 1991.

N. Takayama. *An Approach to the Zero Recognition Problem by Buchberger's Algorithm.* Dept. of Mathematics, Kobe University, submitted to publication.

S. Wolfram. *Mathematica: A System for Doing Mathematics by Computer.* Addison - Wesley Publishing Company, Redwood City, California, 1988.