FRPOLY: A Benchmark Revisited

Richard J. Fateman
University of California at Berkeley

Abstract

The FRPOLY Lisp performance benchmark [3] was based on a circa-1968 piece of code for computing powers of polynomials. We address two questions: (a) What algorithm would you use if you really wanted to compute powers of polynomials fast? and (b) Given that Common Lisp supports many types of data structures other than the simple lists used for the benchmark, what more efficient representations might be appropriate to use for polynomials?

1 The origin of the original benchmark

The FRPOLY Lisp benchmark was devised by the author to test various Lisp systems on computations typical for a symbolic algebra system[1] – at least one that is written in Lisp. The name for the benchmark combined the dialect in which it was originally submitted (Franz Lisp), and the nature of the computation – polynomial arithmetic. The code originated in a polynomial arithmetic package written in about 1968 by MIT Prof. William A. Martin for use in the Macsyma[2] computer algebra system. Although major and minor variations on representations and algorithms have appeared in the programming literature[3], Martin's design still seems to be fairly sturdy. A good deal of the algorithmic portion of Macsyma depends directly or indirectly on this code.

Martin tuned his code in various ways to minimize the number of `cons`-cells required for temporary storage. When the code was written for the 1.2 megabyte (256k word) address-space PDP-6, the MIT Macsyma group initially viewed the program space as huge – which it was, compared to the previous limits encountered on a 32k word IBM 7090. Therefore code size was considered a secondary consideration[4] compared to speed.

[1]More information on this general area is readily available [1].

[2]A trademark of Symbolics Inc.

[3]Some discussion of the design criteria may be found in the literature on symbolic mathematical computation (e.g. [1]).

[4]In fact, the Macsyma code exceeded that address space within 3 years, leading to schemes utilizing load-on-command files.

Some modifications were made to the low level polynomial code by the author in the course of his doctoral work (prior to 1971), as well as the more recent adaptation into the form of a benchmark. Numerous cosmetic changes could have been made to improve readability, modernize the use of the language (avoiding global variables, using abstraction, even inserting comments, etc.), and in other ways change the somewhat spartan code. Although some obviously irrelevant code was removed[5] none of the other alterations were done.

Changes for compatibility with Common Lisp included addition of proclaim/declare code, and minor syntactic variations such as substitution of "+" for "plus" for the addition function.

2 The point of the original benchmark

For benchmark testing we proposed timing several sequences of computations. Each test case was based on a (well-known to be non-optimal) algorithm for computing powers of polynomials ("expanded out") where the polynomials were represented in a so-called sparse recursive linked-list form. For example, $9x^2+7$ would be the list `(x 2 9 0 7)`. This could be read as x power 2 coefficient 9 plus power 0 coefficient 7. In the same representation, $x+y$ would be `(y 1 1 0 (x 1 1))` or `(x 1 1 0 (y 1 1))`, depending upon the ordering of the variables x and y in some global scheme. Because this form makes it cheap to represent a sparse polynomial such as $x^{100}+1$ (as `(x 100 1 0 1)`) and it is "recursive" in the coefficients, it is sometimes called a sparse multivariate recursive polynomial representation.

The polynomials proposed were all variations on $p = 1 + x + y + z$. The first was p itself, the second was a "bignum" version with $p_2 = 100,000p$, and the third was $p_3 = 1.0p$ (that is, convert all coefficients to the floating-point (single-precision) 1.0.)

Computing a fixed power of each of these polynomials gives a variation on the measurement. For $p = 1+x+y+z$: Computing p^{15} should give an indication of how well the basic Lisp system performs in function calling, creation of smallish numbers (but necessarily allowing for the possibility that the numbers might be bignums), and to a small extent, arithmetic itself. The intention was to get a grasp of performance on a typical mix of operations. Lisp systems with "immediate number" representations large enough to store most of the numbers in this computation tend to benefit substantially on this test.

For $p_2 = 100000p$: Computing p_2^{15} gives an indication of how fast multiple-word arithmetic can be performed by the Lisp system. By subtracting off the time for the previous test, the cost of the bignum arithmetic is isolated.

[5]Code which dealt with arithmetic on algebraic numbers was excised. A symbol representing an algebraic number such as $\alpha = \pm\sqrt{2}$ has to be treated differently from an ordinary indeterminate, since the relationship $\alpha^2 = 2$ affects the result of multiplying polynomials.

For $p_3 = 1.0p$: the same test gives an indication of the costs of using floating-point arithmetic.

The benchmark requires that the same code be used for each of the tests. The intention was to make it impossible for a benchmark hacker to avoid the testing of the types of numbers at each arithmetic operation[6].

Although one is ordinarily not allowed to change algorithms in benchmarking, it is natural to wonder if the procedure given to compute the 15th power of a polynomial is really so fast. The powering algorithm as given in the benchmark proceeds by successively computing $p^2 := p \cdot p$, $p^3 := p \cdot p^2$, $p^4 := p^2 \cdot p^2$, $p^7 := p^3 \cdot p^4$, $p^8 := p^4 \cdot p^4$, and finally $p^{15} := p^7 \cdot p^8$. On one system we tried (Allegro CL, MIPS M/120 computer), using "low safety" (safety 0) and "high-speed" (speed 3) settings for the code generated by the compiler, and using appropriate declarations, this took about 350 milliseconds.

Another algorithm that might seem to be more costly computes the sequence $p^2 := p \cdot p$, $p^3 := p \cdot p^2$, $p^4 := p \cdot p^3$, ..., $p^{15} := p \cdot p^{14}$. It can be shown than under many circumstances this sequence takes less time (see [2], [4]). Although it uses 14 *polynomial* multiplications instead of 6, the total cost is lower in terms of *coefficient* operations.

By substituting this algorithm for the powering algorithm in the benchmark code, we found the same problem took about half the time – only 200 milliseconds.

Yet this is not the fastest way of computing the general power of a sparse polynomial. In fact, by using a variation of binomial expansion, the same problem can be solved rather faster. The program that has been in Macsyma for 20 years can be inserted into the FRPOLY benchmark, and it turns in the rather remarkable time of 55 milliseconds – some 6 times faster. Variations are described in great detail by Probst and Alagar [4]. The version in Macsyma operates as follows: Consider computing the power of a polynomial $(x + y + z + \cdots)^p$ as $(a + b)^p$ where $a = x$ and $b = y + z + \cdots$. That is, begin by splitting the polynomial into two pieces, unevenly (Pragmatically speaking, it seemed simpler to do this with the Lisp representation than to compute the length of the list, and then evenly count out half the polynomial's terms into each of a and b). Then use the binomial theorem to compute the pth power: $(a+b)^p = a^p + pa^{p-1}b + \cdots + b^p$. In computing the powers of a and b Macsyma uses the sparse powering algorithm for b^2, but then multiplies b^{n-1} by b to get b^n. Adding the parts together when there are repeated monomials whose coefficients must be combined or sorted in order, naturally affects the cost.

Conclusion: even assuming the multiplication and addition programs programs in FRPOLY are fairly good, the powering program should not be taken

[6]In principle one could expand out multiple versions of the code dispatched with type-tests at the top, and special-case in-line code at the bottom. This would be rather devious for benchmarking, but might be quite reasonable for production code.

too seriously, except as a benchmark driver to set up a bunch of polynomial multiplication problems[7].

3 Lisp and compiler declarations, new representations

The FRPOLY benchmark was based on code that predates Common Lisp. It is natural to ask how changes in technology have altered the benchmark.

3.1 Compilers

In the earliest (1968) Maclisp system on which Macsyma was implemented, compiler declarations were used for declaration of "special" variables, and not much else. A revision of that system provided for better compilation of explicitly declared numerical (fixed and floating-point) code. That was still far less elaborate than provisions in Common Lisp today.

Our re-implementation of Maclisp for the DEC VAX computer, Franz Lisp, omitted much of the fast-number compilation mechanism, leaving only special forms for common arithmetic for special cases (such as fixnum plus). It was our view that only a limited set of operations in Macsyma could be *assured* to be restricted to (say) fixnums. Far more frequently one would have to write code that needed to be valid over the integers (including bignums). In-line floating-point code was also deemed less important because we expected that linkages to foreign-function (e.g. Fortran or C) subroutines would take care of most intensive numerical work. By maintaining compatible data structures (Fortranish arrays) we hoped to facilitate the use of foreign functions.

The search for speed took another turn: Franz Lisp provided declarations to restrict the mutability of function calls. That is, with certain declarations in place at compile-time, the subsequent redefinition of functions at run-time would have no effect on calls from compiled code. This was more elaborate than the Maclisp model (the `nouuo` business for those familiar with Maclisp). The major payoff occurred when it was possible to avoid the relatively expensive and general subroutine call instruction on the DEC VAX computer. In practice this technique of using local functions was fairly effective in reducing calling time between user-defined compiled functions.

On the other hand, the representation of small numbers in Franz Lisp was probably less effective than the immediate-number schemes commonly used today.

[7]Strictly speaking, this statement is a bit too strong since there are other classes of polynomials and variations of polynomial multiplication routines for which the FRPOLY powering routine looks better.

3.2 Alternative representations

Although various Lisp systems have provided alternative data types to lists, it has generally been the case that the code generated by compilers for dealing with these representations was not as efficient as could be implemented. Arrays, "hunks"[8], and vectors are plausible alternatives for lists, given the highly regular structure of polynomials. It is also plausible to write programs relatively divorced from the data representation by using "objects" or "structures".

Given the emergence of Common Lisp in several fairly robust versions, it seemed time to try out alternative representations that are supported directly.

We've experimented with several representations within Common Lisp, but of the ones we've tried, it has seemed most appropriate to use simple-vectors for polynomials. The representation we've used is different conceptually from the FRPOLY choice made in 1968. The FRPOLY representation omits all zero coefficients, but must explicitly store all exponents. This is good for sparse polynomials, and is rather advantageous for computing powers of p above.

By convention in symbolic mathematics programs, a polynomial representation that stores all coefficients, even those that are zero, explicitly, is referred to as *dense*. By contrast, a sparse representation such as the one used by FRPOLY stores exponent-coefficient pairs.

For a multivariate polynomial, a completely dense representation explicitly stores each coefficient up to the highest possible power of each variable anywhere in the polynomial, and the exponents are implicit. In practice, such a v-dimensional array is likely to have a large number of zeros, and is rarely of interest, at least for serial computers.

Our representation is a compromise in that it is dense on a per-variable basis, but instead of storing a full v-dimensional array, it stores (in effect) vectors of vectors (recursively). That is, a polynomial is a vector with a main variable, whose subsequent entries are the coefficients of successive exponents, up to some highest non-zero coefficient. The exponents are not stored. The coefficients are some numeric type (for example, integers, single-floats) or, recursively, polynomials in other variables. For example, $9x^2 + 7$ would be the vector `#(x 7 0 9)` and $x + y$ would be `#(y #(x 0 1) 1)` or `#(x #(y 0 1) 1)`, depending upon the ordering of the variables x and y in some global scheme. This form makes it cheap to represent a dense polynomial such as $x^2 + x + 1$ (as `#(x 1 1 1)`), although it is wasteful on $x^{100} + 1$. Similar to the list form, it too is "recursive" in the coefficients.

How fast is this representation? Clearly it depends on the relative speed of referencing simple-vectors versus following lists: this in turn depends on the efficiency of the language implementation as well as the underlying hardware. Vectors are particularly attractive for parallel or pipelined processing, and one

[8]Hunks are blocks of storage similar to arrays, but available only in sizes that are powers-of-two. Hunks are defined in Maclisp and Franz Lisp. Presumably the rationale for them (reduced storage fragmentation) was insufficient to justify their inclusion in Common Lisp.

can, in some cases, make excellent use of machine architectures oriented toward a Fortran-like model of programming. The speed of programs using this representation is also strongly affected by the choice of problem. The FRPOLY benchmark is definitely *not* the best choice for this representation, but it is important that the representation and the algorithms perform well, nevertheless, even on problems for which it is not optimal. An example which is multivariate and dense, and which shows vector representation to greater advantage is the computation of powers of $q = (1 + x)(1 + y)(1 + z)$.

In order to make some judgments, we ran computations equivalent to the FRPOLY benchmark using vectors on a variety of hardware and software systems. Ultimately we found a lack of consistency comparing the various hardware-software combinations available to us. Using vectors for computing p^{15} was 10% to 15% faster than lists on some systems, yet on others, slightly slower. The binomial-expansion algorithm seemed to be more efficiently implemented in lists than vectors, consistent with our belief that chopping polynomials into pieces required more copying with vectors than with (destructive) list operations.

The data size for vectors is a good deal smaller: assuming the coefficients are "immediate" data, the vectors use as little as 25% of the space used by lists. In the FRPOLY benchmark representation, half of the space is used to store exponents. Of the remainder, half of that is used to store pointers. The vector version basically stores only coefficients. If the coefficients are mostly zero, the vector representation is relatively wasteful, but the vector version has a space advantage as long as at least 1/4 of the possible coefficients are non-zero[9].

Conclusion: it is plausible to use vectors for polynomial calculations[10]. There are small constant speed factors favoring vectors, even in the FRPOLY benchmark, in some Common Lisp and hardware combinations.

4 How does this compare to C?

Given the "common wisdom" that programs written in the C language run faster than programs written in Lisp, we tried running the same problem (expanding p^{15}) on several commercially available computer algebra systems written in C. To be somewhat consistent on hardware, we compared data only on a particular model Sun Microsystems 3/50. On that system our fastest (binomial expansion powering) version written in Common Lisp (Allegro CL 3.1.4) takes only 0.7 seconds for a list-based algorithm and 0.95 seconds for a vector-based algorithm.

[9]We are neglecting the 1-word vector header. Including it would not be a significant percentage penalty except for very small polynomials, and operating on them is fairly inexpensive anyway.

[10]To those who object that the vectors are hard to read or write, we can add a parser and pretty-printer. Of course lists can be just as hard to read, and the same remedies apply.

What about the C-based systems, then? Excluding time for display and keeping command-parse time to a minimum (insignificant in these numbers), we found that Maple 4.3 takes 1.85 seconds. It uses an algorithm based on the "balanced binomial theorem" (splitting a multinomial evenly). We also tried Mathematica (version 1.1), a system that uses a representation reminiscent of Lisp prefix trees. The algorithm is not described in the documentation. In that system, the expansion of p^{15} takes 65 seconds. Mathematica version 1.2 reduces the time to 25 seconds.

Among other computer algebra systems, we also can provide Lisp-based times: Using the PSL version of Lisp and the latest version of Reduce, A.C. Hearn reports that the problem takes 4.3 seconds. This is using the Reduce representation and an algorithm that look like the one using "repeated squaring" (as in FRPOLY), rather than one of the faster ones for this problem. If a factor of 6 could be obtained by Reduce by the use of a different powering algorithm, the time would be reduced to about 0.7 seconds.

In other respects, is it realistic to try to write FRPOLY in C? Two factors stand in the way: implementation of arbitrary precision integer arithmetic, and storage allocation.

It would undoubtedly be somewhat painful to write this code in C if it required original code for efficient solution of both these tasks. In the context of a symbolic manipulation system written in C, the cost of developing this code would be amortized among its many applications. Obviously Maple and Mathematica have done so, but the code is not available for inspection.

Although we have excluded "garbage collection" time for the Lisp algorithms, we found that modern "generation scavenging" techniques seemed to reduce the GC time for problems of this size to a very small additional percentage of the total time used (perhaps 10%). Maple has a garbage collector which is fairly unobtrusive—we've excluded that time as well; Mathematica uses reference counts, and we were unable to exclude that storage allocation time. Clearly Mathematica has other problems with this benchmark that are not allocation-based. Perhaps a better implementation of its `Expand` command – possibly softening the commitment to a single prefix-tree representation – would be worthwhile.

5 Conclusions

- The code in the FRPOLY benchmark for polynomial addition and multiplication, combined with a more realistic powering program, is quite fast for the task of polynomial powering (if we agree that we must provide for the possibility of coefficients becoming "bignums"). With appropriate declarations and a reasonable compiler, it compares favorably with code written in C. In particular, the code we tested is somewhat faster than Maple 4.3 and considerably faster than Mathematica 1.1 or 1.2.

Thus it is not necessarily the case that C code runs faster than Lisp code.

- Using vectors in Common Lisp saves space, and perhaps time, over lists. The speed advantage to using vectors is negligible or non-existent for the relatively sparse example used in this benchmark, although both software and hardware systems affect the trade-offs. However, it seems plausible that we should consider using such data structures more often.

  Note that Common Lisp provides a variety of syntactic sugar relating to structures or other objects that can make the use of vectors quite simple. It is not necessary to convert all programs to use vectors – Macsyma uses prefix trees for some purposes in addition to using the Lisp list form used in FRPOLY. However, given the wealth of representation possibilities, a programmer can consider other forms (including prefix trees or hash tables), and use these representations for efficiency. Since the translation to and from vector form can be done in time linear in the size of an expression, a non-linear polynomial operation (multiplication) is likely to dominate the conversion time.

- Although we have not explicitly discussed it here, the ease of coding, profiling, and debugging in Lisp has costs reflected in the size of the run-time package, but not necessarily in the efficiency of the resulting code. The underlying support system, providing storage allocation, arbitrary precision arithmetic, and numerous facilities, is helpful in keeping the user-written code for computer algebra brief and to the point. While current Common Lisp systems seem bloated relative to the minimum set of facilities needed for polynomial manipulation, it is reasonable that "run-time subsets" could be much smaller. If such subsets were small enough, the use of Lisp for production code for computer algebra systems would seem to have no substantial size or speed penalty.

6 Acknowledgments

This work was supported in part by the National Science Foundation under grants CCR-8812843 and CDAA-8722788, the Defense Advanced Research Projects Agency (DoD) ARPA order #4871, monitored by Space & Naval Warfare Systems Command under contract N00039-84-C-0089, the IBM Corporation, a matching grant from the State of California MICRO program, and Sun Microsystems Inc.

Thanks to Gregg Fee of the University of Waterloo for running some tests on Maple, Berkeley students Peter Klier and Tak Yan for programming assistance, Duane Rettig of Franz Inc. for suggestions on "optimization" in Allegro

Common Lisp, and Peter Norvig, Paul Hilfinger and the referees for various suggestions on an earlier draft of this paper.

Copies of the modifications of FRPOLY programs can be requested from fateman@Berkeley.EDU.

References

[1] J. Davenport, Y. Siret, and E. Tournier. *Computer algebra systems and algorithms for algebraic computation.* Academic Press, 1988.

[2] Richard J. Fateman. “On the computation of powers of sparse polynomials,” *Studies in Applied Math.,* (53), no. 2, June 1974, 145-155.

[3] Richard P. Gabriel. *Performance and evaluation of Lisp systems*, MIT Press, 1985.

[4] David K. Probst and Vangalur S. Alagar. “A family of algorithms for powering sparse polynomials,” *SIAM J. Comput.* (8), no. 4, (1979), 626-644. Corrigendum, *SIAM J. Comput.* (9) no. 2, (1980) 439.