Importing the Gnu Multiple Precision Package (GMP) into Lisp, and implications for Functional Programming

Richard J. Fateman
University of California at Berkeley

August 26, 2003

Abstract

Advocating the use of a Common Lisp as a central organizing environment for building scientific computing systems runs counter to the conventional wisdom, which suggests that languages like Fortran, C++, or even Java, are more suitable. We prefer Lisp based on its debugging features, interactivity, memory model, existing code-base for computer algebra and user interfaces, and for purposes of this paper, its ability to dynamically load modules written in other languages.

GMP is an open-source highly-tuned library of programs for arbitrary precision integer, rational and floating-point arithmetic. It is easy to use it from Common Lisp in some ways but used in its most efficient form GMP conflicts with the dominant (functional) model of arithmetic in Lisp and similar languages. GMP may also require a delicate touch to interface with the storage allocation mechanisms in its host language (Lisp's garbage collector). We discuss both these issues.

1 Introduction

We have written on integrating non-Lisp libraries into Lisp previously [1], both by converting Fortran into Lisp, and by linking Lisp to existing (non-Lisp) packages. We would not write about it again except for four factors.

- We have come to a better solution to storage problems.
- The GMP package is a free, impressively reliable, and painstakingly programmed solution to a problem, continually updated to match new hardware[1]. It seems one should re-use it, if its specifications meet your needs; you are not likely to write a system that is better without great effort.

[1] Although it is not necessarily easy to recompile or install on a particular computer system, especially if you are using a proprietary operating system.

- Other languages have adopted a similar integrative role specifically with respect to GMP (Python, C++, Tcl, Perl, Java, Delphi, as well as several computer algebra systems), and many have not done as well in providing full access to GMP as we demonstrate. They therefore may learn from our solution in Lisp.
- Previous papers on this topic are not quite in the open literature, having appeared only in poster sessions and on web sites.

As a brief reminder of why we are using Lisp, let us quote from [1]:

- It has an interactive base, supportive of debugging.
- It has a compiler which is integrated with the system, so that any modules can be compiled or run interpretively.
- Although the base language is rather simple, it has been extended to provide many advanced language features.
- There is thorough support for object-oriented programming. CLOS (the Common Lisp Object System) has probably the most versatile object model of any language. The Meta Object Protocol allows the programmer to define new models making other tradeoffs between compile-time and runtime. This system can ease the burden of extending existing operations to new data types.
- Lisp is known for the general ease with which one can build new languages (most compilers have what amounts to a private version of a mini-Lisp inside them!).
- Common Lisp is an ANSI Standard language.

2 Why GMP?

GMP is a versatile multiple-precision arithmetic package supporting integers, rationals, bigfloats. This in and of itself is of no special interest since Common Lisp supports integers, rationals, and several floating formats (the standard does not require that they be distinct, but usually single and double are implemented), and there are several bigfloat packages written entirely in Lisp.

Yet GMP has a following that suggests it is going to be better than the corresponding facilities in any ordinary Lisp: a coterie of fanatics familiar with different architectures is constantly on the prowl for hacks to make it faster. There is a range of domains from the just-larger-than-double to the truly huge over which GMP is either the fastest of the (many) competing big-number libraries, or nearly the fastest.

GMP supports certain kinds of operations not usual in Lisp or other functional programming languages, that used carefully can make programs more efficient. In particular, in-place arithmetic is potentially a big time saver.

Additionally, the source code for GMP is free and available for essentially any contemporary architecture and operating system.

The downside of GMP is that of any system that allocates its own storage: it does not know when that storage can be returned, and thus relies on some other mechanism. Some languages provide no mechanism and hope that you simply have enough storage to run to completion. Others must do explicit freeing of storage based on some calculation, perhaps based on static scope and variable bindings. Others may do reference counts. In our case we use Lisp garbage collections.

3 How we use GMP

There are two models we used in a recent paper on fast polynomial multiplication.

3.1 Batch Allocation and Deallocation

The first version of our multiplication program stored all the non-zero coefficients in the answer in a hashtable. This hashtable grew to the necessary size, and perhaps somewhat larger: that is, some terms can be reduced to zero as in multiplying $(x-1)(x+1) = x^2 - 1$ where the coefficient of x becomes zero. More often the same coefficient is repeatedly updated. This is great for GMP, which has as a "unit" operation `x:=x+a*b`. At the end of the multiplication, all the non-zero coefficients can be copied to another data-structure (or left in the hashtable), and the zero coefficients explicitly removed (returned to the GMP allocation).

This works quite well, but provides no sharing, as is commonly used in some computer algebra systems, especially those using a recursive representation of polynomials rather than the expanded version implicit in the hashtable representation.

3.2 Interactive Allocation and Deallocation

Ordinarily one would like a GMP number to exist only as long as it is accessible, and otherwise deallocate it. GMP does not provide much help with garbage collection. Here is what it could do, in principle.

First, realize that a GMP integer number is initially an array of three 32-bit quantities. A maximum length M in "limbs" (a limb is what we usually have referred to as a "big digit" or bigit of which the number is composed). Typically this means M words are allocated for this number. The second quantity is the actual length N in limbs that this number occupies (less than or equal to M), and a pointer P to the address of the N limbs. The actual length is multiplied by the sign of the integer. If the number grows to need more limbs, a new section of memory is allocated and pointed to. If the number shrinks, the memory is *not* deallocated, except by explicit program request.

Other GMP quantities are built out of these pieces.

GMP allows the user of the library to provide an allocation/deallocation mechanism for the allocation, and when a number grows, presumably allocates the new memory, copies appropriately into it from the old memory, and deallocated the old.

We are uncomfortable providing Lisp storage for GMP to use because Lisp storage in some implementations has the annoying property of possibly moving after it is allocated. That is, in a copying garbage collector, an array may move. We do not want GMP to have a pointer to an array in a register and trigger a GC by some allocation.

A way around this is to disable GC during GMP internal operations or to allocate Lisp arrays in static space that is not copied or GC'd.

We considered using this, nevertheless, until we hit upon another strategy.

The three-word initial allocation L of a GMP number is an ordinary Lisp datum: an array of 3 unsigned 32-bit quantities, none of which are pointers into Lisp space. If there are no pointers to L, then L itself will be garbage collected. But the storage pointed to by P is no longer in use, and we must tell GMP to deallocate it, too.

This is the simple trick, available in some Lisps (but not required by the ANSI Common Lisp specification). We use it in Allegro CL, but an identical extension is available in Harlequin/Liquid Common Lisp.

We attach to the length-3 array L a `finalization`. This is a program to be run when the garbage collector determines that L is inaccessible[2]. The Lisp system runs the finalization procedure. In our case it calls the GMP deallocator (the GMP function `mpz_clear`). This action also removes or unschedules the finalization. This, in effect, delays the actual collection of L by one more GC, but L as well as the GMP number it contained both get collected.

A minor hack on this is to make the finalization also put the now "empty" GMP number on a list of available "carcasses" of GMP numbers to be re-used: the array L is then popped off this list and used when next there is a need for a GMP number. L is re-initialized and provided as workspace. If more GMP numbers are needed, they are produced as necessary. If the list of available carcasses seems too long, it can be reduced and the carcasses GC'd. It would be reasonable to do a global GC at such a time (in a generation-scavenging GC). In this case the finalization we use does two things. It sets the storage used by the GMP number to be just enough to store the number 0, and then it saves it on a stack. `(mpz_set_si x 0) (vector-push-extend x *gmp-free-list*)`.

Yet another improvement is plausible and we would recommended it in languages which support it. (It would require some lower-level coding in Allegro CL). If all of the 3-element arrays were typed as GMP-headers, and GC of a GMP-header caused a deallocation in the specified way, then the interaction of the GC and the GMP system would be about as efficient as possible. Conceptually we would be attaching a finalization to the CLASS of GMP numbers rather than each number.

[2]except from the list of objects to be erased that have pending finalizations

4 Alternatives to GMP for bigfloats

There are numerous alternatives, some listed in the references. For a comparison of systems, see Dan Bernstein's periodically-updated web site `http://cr.yp.to/speed/mult.html`. Bernstein's benchmarks illustrate that GMP is hard to beat as a general facility. Bailey [2] Wyatt [16], Brent [5], Fateman [7], Sasaki [13], Pinkert [12], and systems such as `dc` in the UNIX system [14], or components of Maple [6], Mathematica [15], PARI [4] and Saclib [10]. See also LiDIA [11] Haible's CLN [9].

5 Strategies for Implementation

Links to the GMP library from Allegro CL are easy to establish. For example, in the Windows world one essentially loads a single "dll" containing GMP version 4.1. For each subroutine of interest one declares the foreign-function types of arguments and return values. The novelty in Lisp or functional languages is that the routines usually deposit their results in a target location. If one implements (as is the case for GMP and Python) a functional version of `x:=a op b` by ALWAYS computing a new result for `a op b` and then storing it in `x`, one has a rather inefficient version of `x:=a op x`. Thus we make available both a functional version and an imperative version in Lisp. See [1] for details.

6 Remaining issues

Error handling (such as GMP running out of space) are not, but could be handled more gracefully. Perhaps an integration with Not-a-Number concepts from the IEEE 754 binary floating-point standard could be used.

7 Acknowledgments

This research was supported in part by NSF grant CCR-9901933 administered through the Electronics Research Laboratory, University of California, Berkeley.

References

[1] R. Fateman "Importing Pre-packaged Software into Lisp: Experience with Arbitrary-Precision Floating-Point Numbers" Poster session ISSAC 2000, St. Andrews, Scotland UK. See `www.cs.berkeley.edu/~fateman/papers/floatprog.pdf`

[2] David H. Bailey. "MPFUN: A Portable High Performance Multiprecision Package." NAS Applied Research Office, NASA Ames Research Center,

Moffett Field, CA 94035. March, 1991. A report was published as "Algorithm 719, Multiprecision translation and execution of FORTRAN programs." *ACM Trans. Math. Softw. 19*, no 3 (Sept, 1993) 288—320.

[3] David H. Bailey. "A Fortran 90-based multiprecision system." *ACM Trans. Math. Softw., 21* no 4, (Dec. 1995) 379—87. available at `http://www.netlib.org/mpfun/`

[4] C. Batut, D. Bernardi, H. Cohen, M. Olivier. User's Guide to PARI-GP. Feb, 1991 (version 1.35).

[5] Richard P. Brent. A Fortran Multiple Precision Arithmetic Package, *ACM Trans. Math. Softw. 4* (March, 1978), p. 57 – 70.

[6] B. Char et al. *Maple Reference Manual*, Springer-Verlag, 1992.

[7] Fateman, R. 'The MACSYMA 'Big-Floating-Point' Arithmetic System," in: R. D. Jenks, (ed.), *Proc. of the 1976 ACM Symp. on Symbolic and Algebraic Computation,* (SYMSAC-76), Yorktown Height, N.Y., August 1976, 209-213.

[8] R. Fateman, K. Broughan, D. Willcock, D. Rettig, "Fast Floating-Point Computation in Lisp" *ACM Trans. Math. Softw. 21* no 1. March 1996 26–62.

[9] Haible, Bruno. CLN `http://clisp.cons.org/ haible/packages-cln.html`

[10] Krandick, Johnson. SACLIB. RISC-Linz

[11] `http://www.informatik.th-darmstadt.de/TI/LiDiA/`

[12] Pinkert, J. R. "SAC-1 Variable Precision Floating Point Arithmetic," *Proc. ACM 75*, (1975) 274–276.

[13] Sasaki,T. "An Arbitrary Precision Real Arithmetic Package in REDUCE" in *Proc. EUROSAM 1979, Lecture Notes in Computer Science, 72,* 1979, Springer-Verlag pp. 358–368.

[14] UNIX manual.

[15] Stephen Wolfram. *Mathematica – A System for doing Mathematics.* 2nd edition. Addison Wesley, 1990.

[16] W. T. Wyatt Jr., D. W. Lozier, and D. J. Orsen. A Portable Extended-Precision "Arithmetic Package and Library with Fortran Precompiler," Math. Software II, Purdue Univ., May 1974.