A Revisionist Approach to Algebraic Expressions in Lisp.

Richard Fateman
Computer Science Division
Electrical Engineering and Computer Sciences
University of California, Berkeley

December 3, 2010

Abstract

The design of Lisp by John McCarthy was influenced by a prototypical application of computing symbolic derivatives of algebraic expressions. Ever since then there have been computer algebra systems (CAS) which used Lisp as a base language, and used this initial design for a start.

As CAS have become more ambitious, aspects of this initial design came to the fore, suggesting that it didn't scale up. Some programmers have asserted that "better languages" are needed for CAS. We review what has happened, and how a more modern (but still Lisp) approach to the problem emerges.

1 Review

First a review of what the how Lisp's initial design was influenced by computer algebra.

We somewhat simplify the situation for ease of presentation.

An algebraic expression (AE) is either a symbol like `x, y` or `z`, an integer, or a list consisting of an operator followed by 0 or more AEs. The operator is typically chosen from a rather small collection, but even if it is arbitrarily extended to "user defined" operators, for CAS purposes, the most prominent operators are `+, *, /, ^` . These stand for addition, multiplication, division, and "raising to a power". Addition and multiplication being "n-ary" as well as the most common, must be dealt with especially well. In most CAS there are other operators which cause fewer problems, partly because they have a fixed number of arguments:`log, exp, sin, cos`[1]....

One of the most important and yet not entirely well understood problems in a general-purpose CAS is that of simplification. Simple rules like converting $x+x$ to $2x$, or in Lisp, transforming `(+ x x)` to `(* 2 x)` can be programmed in various ways[2].

Now consider a very long expression r=`(+ .... (* 2 x) ....)`, and consider the task of simplifying `(+` r `x)`. One way to do this is to search down the list r looking for the only term that includes a pure `x` term, and combine the other input, `x` with it, producing a new expression r'=`(+ .... (* 3 x) ....)`.

If you have not thought about this problem, consider the complications that might ensue if the list r instead contains terms that look like `(* 4 x y)` or `(* 5 (^ x 2))` or `(* -1 x)` or `(* 3 (+ x y))`. Three of these terms do not combine with the `x`. One of them "cancels" the `x`.

If we are computing the simplification of `(+` r `(+ (* 7 x)(* 8 (^ z 2))))` we must also note that `(* 7 x)` combines with other terms which are multiplies of `x`, not just `(* 7 x)`.

If we see AE like `(+ (* 3 x y)(* 3 x))` we might wish to produce the expression `(+ (* 3 x (+ y 1)))`, though there are generally too many grouping alternatives to make such programs obvious.

2 Complexity of Simplification

If r is a list of length n, then finding the term with which to combine `(* 7 x)` will take $O(n)$ time. If we are adding n items one at a time into a list of length n in a sequential approach, we have an $O(n^2)$

[1]The exact appearance of the operator may differ from this. For example, Macsyma uses `(mplus simp)` instead of just `+`.

[2]Data directed, object-oriented, rule-based, functional, procedural,....

operation. This is harmless if n is small, as is often the case. If, as we suggested, r is very long, we may have a performance problem.

Can we re-think the initial design (which is, after all, 45 years old), and come up with a better solution? In fact, there are several approaches, one of which dates back about 35 years to the initial design of Macsyma: use data structures that are much more uniform: keep a canonical ordering of variables and a list of terms by degree, so that merging of lists while adding is easily supported. This is a recursive representation of polynomials in several variables where a main variable is chosen, and coefficients are arranged by degree of the main variable. Each coefficient is either an integer or a representation of a polynomial in variables (of "lower complexity" than the main variable) recursively. The initial "canonical rational expression" form for Macsyma was a list of alternating degrees and coefficients. A similar encoding was used in Reduce and in Matlab68, early CAS also written in Lisp.

Another solution (which we used in "MockMMA" a simulation of Mathematica) is to make a vector of coefficients (the degrees being implicit in the vector index). The coefficients themselves are either integers or (recursively) vectors in other variables. This provides, at least in principle, access to the nth coefficient of a univariate polynomial in time $O(1)$.

This recursive representation works well if there are one or a few variables used to modest degrees. It can require rather deep "diving" into recursive structures if there are (say) v variables and v is a million, and each occurs only once. Then access to a coefficient would be $O(v)$. Another "dimension" in which the vector representation is stressed is if one must store $3x^{10000} + 4$ with all the intervening 0 coefficients explicit; it is just a vector of length 2 if the terms are stored with coefficient + exponent pairs.

Are there storage models with even better performance characteristics? What if we represent a sum like expression r above, as a hash table?

Then searching for a term that can combine with `x` can be done (probably) in time $O(1)$. We can do this by requiring the term $3x$ already in r be stored with a key `x` and a value 3. More generally, a term $5x^2y$ would be stored with a key `(* (^ x 2) y)` and a value 5. (An overall less-attractive alternative design is one where the term $5x^2y$ is be stored with a one-variable key `(^ x 2)` and a value `(* 5 y)`.

3 Implementation hints

There are several tricks that contribute to an efficient implementation. Storing "kernels" like x^2 uniquely means that the Lisp hashtable can be an `eq` table, making the computation of hashcodes faster. Another trick is the support some kind of meaningful ordering of kernels so that we do not have alternative keys in the same hash table that look like `(* a b)` and also `(* b a)`. One CAS, Maple, seems to pick what amounts to a random ordering based on allocation addresses of AEs. A more meaningful ordering (say, alphabetically, and by degree or other complexity measures) is probably desired by most CAS users. Furthermore, in many modern Lisp systems the memory location of an object can be altered by memory allocation operations (a copying garbage collector). While the "address" of a Lisp object can usually be accessed by some implementation-dependent program, it can only be considered "a constant" if that object is specially treated (in some implementation-dependent way) so it is not moved. There are other times when such addresses are needed as for example communicating pointers or array addresses to programs written in non-Lisp languages. These (fairly common) foreign-function interfaces are unfortunately not part of the ANSI standard for Common Lisp.

4 How well does this all work?

Compare the Macsyma CAS before and after these changes.

Construct a list of length n, k=$\{a(1), \cdots, a(n)\}$.
This can be done in Macsyma by `k: makelist(a(i),i,1,n)$`.
Now add these terms together. This can be done by `apply("+",k)$`.

On a 933Mhz Intel Pentium, using Allegro Common Lisp 7.0, for lists of length n=3, performing that *apply* using hashing is about 13 percent slower than the conventional simplifier.

For lists of length 11, hashing is faster by 40 percent. Based on interpolating some data points, the growth in time cost as a function of the number of terms appears to be heavily quadratic, but with a significant cubic component. Extrapolating from test data, we estimate the conventional simplification program on a benchmark of 100,000 terms would take over a day. Using a hash table it takes about 0.5 second.

In Maple 7, another CAS, `k:=seq(a(i),i=0..100000):` followed by `'+'(k):` accomplishes a similar result (unsorted addition of terms), in about the same time. The Maple implementation uses hashing prominently.

If we change the expressions from $a(1)$, $a(2), \cdots$, to distinct symbols like `a1, a2, ...` then Macsyma takes 0.14 seconds, Maple takes 0.43 seconds.

Thus Lisp can be faster than Maple on this extreme test.

5 Conclusions

Lists are a neat representation for algebraic expressions, nicely supported by Lisp for simple algebraic manipulation on modest-sized expressions. Unfortunately, pushing the limits of memory and time for dealing with very large expressions, a computer algebra system today may benefit from alternative data structures. In particular, operations on very long lists may be significantly less efficient than vectors or (as shown here) hash tables. These operations include combination of terms as done in simplification of sums.

Fortunately the change from lists to hash tables can be done without re-writing the whole system. In fact the changes can be incorporated into the same Common Lisp framework, assuming there is an efficient hashtable implementation.

More details as well as references and access to program listings are available in `http://www.cs.berkeley.edu/~fateman/`