Experiments in Hash-coded Algebraic Simplification

Richard J. Fateman
Computer Science Division
Electrical Engineering and Computer Sciences
University of California at Berkeley

August 12, 2005

Abstract

An algebraic expression simplification program is one of the key components in any computer algebra system. It is also among the first non-trivial programs a student learning Lisp may encounter, converting symbolic expressions like (+ x (* y 1) 0) to (+ x y). The solutions expected by the texts' authors are short Lisp demonstration programs but are hardly indicative of the state of the art in computer algebra. In fact the simplest approaches do not scale up, and other ideas are needed to get an efficient simplifier. There are several routes to a more realistic simplifier. Canonical simplification for subclasses of expressions (say, of polynomials) provides great opportunity for efficiency. Even for "general" expressions, the marking of "already simplified" subexpressions is essential. In this paper we explore the idea that a hash table encoding of expressions can provide a way to scale up a "general" simplifier that would otherwise be inefficient on large expressions. We have augmented a relatively traditional Lisp language algebraic expression simplifier from Macsyma/Maxima, using tricks similar to those used in Maple, to achieve comparable or superior performance. This is done without a proprietary kernel or "hacking" with low-level address computation, or writing in lower-level languages.

1 Introduction

Simplifying algebraic expressions is a traditional demonstration application of Lisp programming. By simplification we mean (among other things) transforming $x + x + 3^2 + 5^0 + 1*y + 0 + z*z$ to $z^2 + y + 2x + 10$ or some similarly "reduced" expression. Writing simplifier programs (including parsing the linear input and displaying the result) provides good fodder for student exercises, and can be used to illustrate many aspects of programming techniques [9]. It is, of course, possible to write such programs in other languages, although the typical conventional programming languages such as C++ or Java require substantial additional effort.

Yet most of the solution programs constructed by students in answer to homeworks, or even as part of more ambitious programs, regardless of the implementation details (rule-driven, object-oriented, top-down structured, etc.) have a subtle problem. Since the sample inputs tend to be small, the student does not realize that the programs may take time that is a high power or even exponential in the size of the input.

Not all student solutions are slow, but for the most part a naive program trades off between speed and generality. That is, fast programs are often brittle: they are inadequate on some inputs; nominally "correct" programs with a fairly comprehensive approach become annoyingly slow on moderate problems: perhaps those that can easily fit into today's desktop workstations.

A simple subproblem of general simplification is the problem of simplifying a sum of terms like z, $3x$, $4y^2$, $-z$, and $4x$. To see if terms cancel or combine, the terms must be collected somehow; They could be sorted so that $3x$ and $4x$ are adjacent, and z plus $-z$ cancel. An alternative approach that may be faster is to use a hash table where the keys in our example could be x, y^2 and z.

The balance between speed and completeness takes different forms. One Lisp program (since 1982 a commercial product), Macsyma [7] as well as its free/open-source clone Maxima, has one "general" simplifier with a variety of transformation mechanisms that make it dominate the typical student program on the task of "general algebraic simplification." By keeping track of sub-expressions that are already simplified, it is vastly faster than it would be otherwise.

Yet it can also be far slower than some other systems (among widely-used systems, Maple [1] is often much faster, and seems to have similar capabilities). What's going on?

While there are, in fact, simplifiers within Maxima that are much faster than its general simplifier, these simplifier programs are specialized for domains which direct all expressions into canonical forms. Thus a polynomial can be "multiplied out" and converted canonically into a sequence of its coefficients.[1] Polynomial simplifiers will not solve all problems, and so other programs in Macsyma can be used to efficiently recognize $\sin(x)\cos(x) - 1/2\sin(2x)$ is zero. Yet another will (efficiently in its own way) rewrite the same expression as $1/2(2z_1z_2 - z_3{}^2)$ for appropriate assignments to $\{z_i\}$. Macsyma has no "canonical form" program that recognizes that $x! - x(x-1)!$ is zero, but a program called `minfactorial` attempts to simplify away factorial forms and in this case notices the simplification.

This paper describes some experiments we conducted to determine the effects (on efficiency) of adding more "structure" to the data used by a general simplifier, making Maxima faster, but without losing generality.

In essence, we considered how to providing multiple representations of data – several ways of viewing the same information – where previously only a single view was available. It was our intention to avoid predisposing the results to appear in a specialized canonical form. If we wanted to produce a canonical form, we could do so more directly.

In particular, we use two ideas, both implemented through hash tables, and hence dependent on an efficient implementation of this facility:

- We guarantee "unique" stored-once versions of most[2] data.

- We allowing the tagging and immediate look-up of attributes of expressions as a consequence of their uniqueness, in hash tables. An important attribute of an expression is how it should be ordered with respect to others if they are added together. This ordering can be encoded by placing all relevant objects in a tree, inserted one at a time. Initially this is done with a careful walking through the two expressions being compared. The number of (elaborate) comparisons needed is logarithmic in the number of elements in the tree. When an expressions is finally placed in the tree we can assign the new object a numbering according to its location in the tree, and store that number as part of its hash value. This makes it simple to compute the relative order of objects. Insertions into the tree ordering can always be done by exact rational interpolation, or one could use another numbering scheme.

The data-structure ideas were inspired by (in particular) the Maple computer algebra system, in an attempt to more clearly delineate reasons for the Maple system approach to be speedier. In our view, such a discussion is often muddied by claims that a program's speed is primarily a matter of underlying programming language choice, rather than comparing algorithms or data structures. That is, differences will be explained by claiming that the use of C in Maple's kernel) is "naturally" faster than Lisp (used in Maxima, as well as our experiments). Or in the other direction, that the use of Maple byte-coding for non-kernel programs would inevitably be slower than (compiled) Lisp. We would like to draw attention to the algorithms and data structures, if in fact (as in the examples in this paper) they dominate performance. Programs involved

---

[1]Some computer algebra systems, most notably REDUCE, almost always use a canonical form simplifier for rational functions and therefore are relatively efficient – comparable to Maxima's rational canonical form simplifier. But such a choice makes REDUCE less flexible in some kinds of representation problems.

[2]Two mathematically equal expressions may be stored in separate locations if they are bignums, or when each is stored in a hash table.

in simple manipulations of a linked list or a hashtable or an array should be comparably efficient given "A sufficiently good compiler," for C or Lisp or Java. The language should not matter as much as the data structure.

Our experiments support the idea that a non-canonical simplifier—one that is "conservative" and does not make drastic changes to the form of the expression, a simplifier that can be quite useful and reasonably fast—benefits from hash table data structures in addition to lists and trees. We also show this does not mean it has to be written in C; Lisp has hash table support as well as its traditional rees and lists[3].

Where is this leading? We show below that by making some different choices, we can make the simplifier, still written in Lisp, run hundreds or thousands of times faster than it had run previously.

Our initial goal was to boost performance significantly. In response to reviewers[4], we added a criterion that we should look for comparable or superior performance (vs. Maple, the apparently fastest of the general purpose systems). Also, we should address the question as to whether we can get similar performance while avoiding coding in a low-level language. Reaching this goal relies on the efficiency of the hash table routines of the underlying Lisp. The Lisp system routines may make rather different trade-offs on generality and speed, and consequently may not be as fast as routines specialized to Maple-style data only. Furthermore, the efficiency of these routines vary substantially depending on the Lisp implementation. Given a set of tools we see how far they will carry us. On the other hand, clever people may have hacked on the Lisp system routines for decades.

2   Why Build Experiments?

There are many ways of comparing computer algebra systems. For example, you can see which systems get the right answers to mathematical problems, or see how much prompting each takes to solve a problem [11].

Often systems are compared by timing them on a set of problems. This gives an advantage to the least-sophisticated of the systems just capable of doing the job: it has the least unnecessary baggage. Benchmarks fail to reveal the advantages of untested features.

Yet one area for improvement over a number of systems has emerged from computer algebra system comparisons: it seems that *some systems have made improvement in speed with little or no loss in generality*, by just the kinds of features we have previously mentioned. Incorporating hash coding "canonicalization" of data-structures, Maple [1] seems to have a significant advantage compared to alternative systems of comparable generality.

Macsyma/Maxima is notable in that there is already a multiplicity of data structures. Adding another form, hash coded sums (and products), while hardly trivial, is at least possible. In this effort we build in part on earlier work [10, 3] to use hash coding in the Maxima simplifier [5]. [2].

Since Maxima's kernel source code is open we can rewrite some key parts, as long as we maintain the appropriate interfaces. We can experiment witn Maple (whose proprietary kernel is unavailable to us), only by external measurements[6].

---

[3]There are, as we have noted, also other reasons for using Lisp. There are excellent tools for prototyping, debugging, and large-system building.

[4]Who essentially said that Maple does this already, so who cares if you speed up some other program.

[5]This paper was first drafted in 1991. As we review this paper in 2005 14 years later, one might ask, why are we still using Lisp and not, say, Java? Note that 14 years ago Java didn't matter. 14 years from now one might ask, why not C#? Lisp is 40+ years old and has outlasted, in some ways by mutating, many fads. We are, as already mentioned, partial toward Lisp as a prototyping language. As a base language for symbolic mathematics, so much code has been written in Lisp, including the large systems Scratchpad/Axiom, Reduce, and Maxima, as well as several interface programs under development that it seems plausible to use Lisp for further work as well. Also, as you will see, our test results suggest speed is possible too.

[6]Years ago, some experimental data provided to us on Maple *with hash coding disabled* suggested that this would result in a massive slowdown. But this seemed unfair, since we suspect that a few critical algorithms were changed from linear to exponential cost as a consequence; had they been written without knowledge that hashing was being used, the change from hashing to non-hashing would have been less severe.

By inserting *hash coding and storage of only unique copies of data* we can make minimal global impact on the overall shape of the system; most programs are entirely untouched. Thus we can run the same tests in the same framework. The results should be clearer than comparing the gross timing results of running one computer algebra system against another, or handicapping an existing system by taking away a facility that all the programmers assumed was there and depended upon in their design.

2.1 Adversarial Problems

We consider two obvious *adversarial* problems to a sum-simplifier. That is, the problems are set up to make the simplifier work as hard as possible as a function of the size of the expression being simplified. The first is to pose a case where many terms exist but the ith term is some unique atomic symbol a_i. That is, all efforts to compare and combine terms are fruitless. The second case is where the ith term is an iterated function. For example, term 0 is 0, term 1 is $f(0)$, term 3 is $f(f(f(0)))$ etc.

For the first case the simplifier needs to rapidly ascertain that no two terms are in fact the *same* symbol, and that they are available in the right order by some lexicographic ordering. For the second case, in order for the simplifier to make a decision that two terms are not equal *and are in the right order* requires delving into each term to see which has more f-depth. If they have the same depth they should be combined.[7].

Doing this second task in a naive sequence of insertions resembles was programmed in Macsyma, and resulted in a program running potentially in time exponential in n the number of terms in the input, if terms are repeatedly scanned and sorted into a linked list. A better algorithm using appropriate data-structures may be only $O(n^3)$, $O(n^2)$, or possibly $O(n)$. Of course we do not wish to settle on a design unless it is also efficient on small problems.

2.2 The Approach: Redesign of a Simplifier

We took a functionally closely-specified simplification program (the programs to simplify sums and products) from Maxima, somewhat simplified them (in ways that are inessential to our study), and duplicated the functionality with a design based on hash tables and storage of "unique" versions of simplified expressions.

How can we simplify a sum of terms? For concreteness one might as well imagine a Lisp expression that looks like (`Plus ...`). With appropriate modifications a simplifier for sums might work for any commutative "n-ary" operation: however, the only other operator that is likely to really matter is multiplication. It turns out that the major practical baggage of a simplifier is providing efficient simplification of sums of products. (Dealing with rational and symbolic powers is another touchy area, but there are fewer gross efficiency problems – while a sum of 100 or even 10,000 terms is plausible, a "power" has only one base and one exponent).

We need a program that, given two existing expressions, returns their sum. If we are asked to simplify some unordered collection of terms such as might be typed in by a user of a computer algebra system, we can start with an empty sum and add terms, one at a time.

A sum in "list" form looks like (`Plus` $term_1$ ... $term_n$) where each *term* is of the form

(a) an atom such as `x` or `3`.

(b) a list beginning with `Plus` and containing $n > 0$ terms.

(c) a hash table of type `Plus` and containing $n > 0$ terms. (this item is the major deviation from past practice in Macsyma and is described in detail in the next section.)

---

[7]Since Maple does not normally order terms that are not equal, its simplifier has a substantial advantage in direct comparisons with other simplifiers: it just doesn't do as good a job.

(d) a list beginning with `Times` and containing $n > 0$ *factors,* or (in principle) it could be a Times hashtable. (In practice it is not a hash table because we un-hash it.) The definition of factor is given below.

(e) a list beginning with some other "head" such as `Power` or `Cosine`.

A factor looks like

(a) an atom such as `x` or `3`.

(b) a list beginning with `Plus` and containing $n > 0$ terms.

(c) a hash table of type `Plus` and containing $n > 0$ terms.

(d) a list beginning with some other "head" such as `Power` or `Cosine`.

By convention (and as a result of the simplification process having been previously applied to sub-expressions) we can assume that in sub-expressions which are themselves sums in list form (`Plus ...`) , at most one of the terms is a numeric constant, and if that constant is non-zero, it appears first. Otherwise the constant is absent.

By convention (and as a result of simplification) we can assume that in a product (`Times ...`) at most one of the factors is a numeric constant, and if that constant is not equal to 1, it appears first. Otherwise the constant is absent.

Furthermore, and this is very significant, if we see two expressions with common terms (say, $\sin x$ appears twice), the terms are identical – they are stored in the same address in memory. This makes it possible to *rapidly* determine (in essentially one machine instruction), whether two terms should "hash" to the same address in a table.

2.3   Hash tables

A sum in "hash table" form is the important novelty in our Lisp simplifier design. Novel for Lisp simplifiers in that we don't use the obvious traditional, well-supported data structure: an ordered linked list, or an array[8]. Hash tables are a standard feature in Common Lisp [12]. The advantages to this, in brief,

1. Searching for an appropriate place for an insertion into the table is usually $O(1)$ rather than $O(n)$ where there are n previous elements.

2. By appropriate choice of keys, the hash table organization will automatically merge to the same location elements like $3x^2$ and $5x^2$, both with key x^2 to yield $8x^2$.

3. Elements need not be ordered except in the eventuality that we require displayed "output" from the form. That is, the program need not be concerned with whether (`Plus a b`) or (`Plus b a`) is simplified. [6]

4. If a hash table gets too full, it will be re-allocated a larger space and the elements re-inserted. (This is not a free operation, of course, but it is done automatically by the Lisp system).

In Common Lisp there is a built-in program `make-hash-table` to construct hash tables with various characteristics. It is simple to specify the creation of one with s-expressions (symbolic expressions) as keys as well as values. If we do this with canonical "kernels" we can use `eq`, the Lisp pointer-equality testing function for testing key-equality. ((`make-hash-table :test #'eq`) (One alternative would be to use `equal` for (in effect) graph isomorphism. This is far less efficient, however.)

Given an existing sum, let us assume it is already arranged in hash table form, and let us add another term to it, in each of the cases (a) - (e) above.

---

[8]Lisp has a well-supported array facility.

(a) an atom: if the term is a number, say 9, enter or merge[9] key=1, value=9; if the term is an atom, say X, enter or merge key=X, value=1.

(b) a `Plus`: simplify each term in the list, and then enter or merge it into the main hash table.

(c) a `Plus` hash table: each term will already be simplified. Enter or merge each term.

(d) a `Times`: There are actually two choices here, corresponding to what is usually referred to as "distributed" or "recursive" representation.

For recursive representation, for each term, simplify that term, then find the most complex factor in it. Extract the numerical coefficient if any, from that term and enter or merge the record appropriately as in these examples (assume z is more complex than x or y). If the term is (`Times 45 x y z`) then enter a record key = `z` value = (`Times 45 x y`) If the term is (`Times 120 x`) then enter a record key = `x` value= 120.

For distributed representation, the case of (`Times 45 x y z`) is set up as key = (`Times x y z`) and val = 45. In this representation, every val is a number. (The key could be stored as a hash table or as a product in "simplified" output form.)

The "distributed representation" form given by $45xyz + 3z + 5yz$ is written $((45x+5)y+3)z$ recursively.

(e) Anything else: treat as an atom as in part (a).

We must explain "enter or merge" now. If, in the hash table, there is already an entry with the same key, then the values must be added. For example, if the table has (key=X, value=2) and you enter/merge (key=X, value=3), the result must be a merged entry: (key=X, value=5). For the distributed form, the values added are all numerical constants. For the recursive form, the values added are (in general) expressions which may need simplification. This is programmed by a recursive application of this process. If the merged entry has a zero value, convention would be to remove the hash table entry entirely, although this may be deferred until it matters, say when printing or converting to a list.

To return to a "normal" list form, it is necessary to traverse the hash table (there is a common-lisp function to map over hash table entries, `map-hash`), and and sort the result by some ordering on keys.

Sorting has the potential to be an expensive operation, and it reaches this potential in conventional Macsyma/Maxima. Fortunately it can be trivialized if we have been careful in our construction of keys to make sure they are unique. In this case the first time we use the key in an ordering relation we can compute its order with respect to every previously encountered unique expression and assign it an index number. We can subsequently impose that ordering value on it. Rather than repeatedly (and rather painstakingly) traversing terms to see which comes first lexicographically, as is done in Macsyma's general simplifier, we can use this computed ordering.

2.4 Sorting of Keys

Finding an ordering relation is in one sense trivial. One way, if it need not reflect any human criteria for ordering, is to just use the machine addresses assigned by some system routine, for the expressions. If expressions are kept uniquely stored and are not moved, this will provide a mapping from each expression to some number, perhaps its address in a machine's memory (between 0 and 2^{32} or 2^{64}, for example. Maple uses such a technique, reserving a certain section of the table for pointers to small numbers or sections of big numbers, symbols, compound expressions). The implementation of Lisp we are using has a system routine called `pointer-to-fixnum` that proves such a mapping. Some Lisp implementations, including the one we are using, can move expressions around from time to time during "garbage collection" thereby complicating the situation, but common-lisp `EQ`-hash-tables provide the functionality we need: mapping from expressions

---

[9]described below

to indexes. Although this technique works, it provides a generally unsatisfactory ordering[10], from a human perspective. It appears apparently random, even if it is deterministically dependent on such issues as the order in which expressions are entered into a Lisp system.

In more detail, our method to assert order is to associate with every (simplified) expression, a deliberately computed index which is a rational number (allowing arbitrary interpolation between indices), which is produced at the first opportunity when the expression is compared to any expression. This index corresponds to some concept of global ordering, at least relative to other previously encountered expressions, as computed by a standard Macsyma/Maxima program (called `great`) developed to provide a unique ordering, presumably transitive, over all algebraic expressions that have passed through the Macsyma/Maxima simplifier. In our `uniq`-system, every expression is installed in an ordering hash table which is tagged with the ordering index. Although it is possible to recompute indexes for old expressions in the tree to insert new ones, say if we used integers or floats and needed room in the index space, it is not needed if we use rationals. In fact, floating-point numbers are probably good too, but might require, rarely, a re-numbering.)

2.5 Enhanced Plus-Simplification

We used the same conventions as Macsyma/Maxima's current simplifier: we did not want to slow down or speed up ours by making "improvements". We would, however, like to note that we considered programming some changes in our algorithm for simplification. For example, where Inf represents an infinity, and NaN a "Not a Number" in the IEEE floating-point sense (or perhaps "indeterminate" in mathematical parlance) Inf+Inf = Inf, not 2 Inf. NaN+(anything numerical) = NaN; NaN+(anything symbolic) = NaN. It would presumably be advantageous to use Common Lisp rational numbers instead of "constructed" ones in Macsyma/Maxima.

3 Simplifying Times

Included in our experimental setup is a corresponding program for simplifying products, a modification of the SIMPTIMES program in Macsyma, just as we modified SIMPLUS for sums.

Some of the new SIMPTIMES program follows routinely from the case of `Plus`, but details have to be changed (dealing with `Power`, and application of the distributive law).

A major complication in Macsyma's simplification of products is its consideration of when to apply the distributive law ("expansion"). It seems to us that the right way to do expansion is to use a canonical expansion form for polynomials and avoid using this conservative representation for that task.

One reason for benchmarking sums rather than products is that sums of 1,000 terms or more are not unheard-of, products of such size are relatively unusual. We also expect that the benchmarking of large products would have the same results as large sums.

4 The Experiment and the Results

Here's what we did: Consider an expression (`Plus ...`) that needs re-ordering, grouping of terms, and perhaps recursive recombination of sums. Present it to the Macsyma simplifier, and also present it to the hash table based simplifier. Observe the speed differences. (The memory resources could be observed as well, though this has become less critical as memory costs drop, and typical workstations (in 2005) are outfitted with a minimum of 256 Megabytes of RAM). We tried modifications of Maxima in Allegro Common Lisp 7.0 and compared with Maple version 7. We also tried some tests on Mathematica version 5.0.

---

[10]Even though it is used by default in Maple. Maple provides an option to sort terms, however.

4.1 Some test results

We ran all these tests on a 933Mhz Intel Pentium machine, using Allegro Common Lisp version 7.0 for Maxima.

Construct a fairly long list, $k=\{a(0),\ \cdots,\ a(100000)\}$.

This can be done in Maxima by `k: makelist(a(i),i,0,100000)$`.

Now add these terms together. This can be done by `apply("+",k)$`.

In Maple, `k:=seq(a(i),i=0..100000):` followed by `'+'(k):` accomplishes a similar result (unsorted addition of terms).

In Mathematica, try `k=Table[a[i], i,0,100000]; Timing[Apply[Plus,k];]`.

The time for the modified Macsyma to perform the addition step is about 0.5 seconds. (Maple 7 time: 0.53, Mathematica 5.0 time: 0.19).

If the items are separate symbols, a0, a1, etc. produced by concatenation of strings, then the Macsyma time is 0.14 seconds. (in Maple, either 0.43 seconds if the symbols are constructed by `a||i` or 43 seconds(!) if the symbols are constructed by `cat(a,i)`. In Mathematica, 0.44 seconds.)

Thus the altered Macsyma is, in several cases, faster than the competition.

How successful is this technique for smaller lists which are more likely to be important in every-day computation than these giants? The break-even point for the hashtable method to be superior to the conventional Lisp technique on our benchmark is for lists of length 4 or 5.

Even for lists of length 3, hashing is only about 13 percent slower.

For lists of length 11, hashing is already faster by 40 percent. Based on interpolating some data points, the growth in 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 our benchmark of 100,001 terms would take over a day rather than 0.5 second.

It is perhaps worthwhile to note that it is unlikely to see such large sums—none of whose terms combine— outside of artificial benchmarks. To illustrate how large the 100,000 term sum is, consider that if we print the result out at about 65 characters per line, 50 lines per page, we will consume about 250 pages. This is a large expression, and it is not likely you would be able to do much else with it! On the other hand, we can look at sums in which terms combine. If we use, say, `a(mod(i,10))` and therefore are collecting, in Maxima, multiples of 10 different "kernels", the running times are moderate. The conventional list "slow" method for adding up 100,000 terms is under 3 seconds. The hashtable-based "fast" way is about .29 seconds.

We are continuing to look for inefficiencies or duplications of computations in the Maxima and supporting Lisp code that make dealing with atomic names faster than other expressions. That is, we think the times for expressions like `a(100)` should be about as fast as symbols like `a100`. We suspect that one contribution to Maple's speed that we cannot duplicate without considerable re-working of the Lisp system is that Maple's design allows expressions to stay in the same address for the duration of a run. The Lisp we are using for testing, Allegro CL 7.0, has a generational garbage collection (GC) algorithm. This, along with most other contemporary Lisp GCs is is a copying system, and so data is periodically re-arranged and compacted in memory. This copying requires an extra layer of indirection in finding unique locations for objects. (A possible clumsy way around this is available by using data structures in space that is not run by the GC, e.g. "foreign" space.) An additional consideration may be the somewhat more general nature of data structures in Lisp, requiring a hashing function that is correspondingly more elaborate. A profile of the Lisp execution show that more than half of the CPU cycles are within two functions, PUTHASH and GETHASH, which could perhaps be specialized more. A final thought: we are, in Lisp, using a general facility for unique "cons" cells. A side effect of this is that all equivalent similarly-simplified expressions are stored in the same location in memory. However, we cannot infer that, just because an expression is in the hash table `*uniq-table*`, that it is simplified. In fact, some entries there are fragments of Macsyma expressions like tails of linked lists. The check for "already simplified" is based on looking at a tag on the expression: the `simp` tag in, for example, `((mtimes simp) a b c)`.

In computing a sum into a hash table, it is important to allocate a large enough table; otherwise re-allocation and re-hashing is required, perhaps multiple times. Fortunately, a simple heuristic is available for sums: we can compute the maximum number of entries theoretically needed by the counting the number of arguments to "+". Thus we can allocate enough so that re-sizing is unlikely: we can even make sure that the table is not more than 3/4 full. A slightly more sophisticated heuristic would consider if any of the arguments to "+" are hash tables themselves, in which case the size of such tables should be added to the count.

As noted above, we encountered some strange anomalies in Maple version 7 when creating new symbols e.g. $z123$ rather than "functions" like $z(123)$ the times became drastically longer. This is apparently similar in later Maple versions.

5   Conclusion

Simplifying using hash tables (i.e. inserting and collecting similar terms, *avoiding sorting the results* [6], and only *if necessary* presenting the answer back as a list) provides the potential for significant speed-up on very large multivariable expressions compared to performance in a computer algebra system not already using these ideas. In our tests, a retro-fitted Macsyma system became competitive with Maple and Mathematica, systems designed ten or twenty years later. The idea is sufficiently effective in our tests to recommend it even for fairly small expressions.

In our experiments using an ANSI Standard Common Lisp, optimizing parameters for re-sizing of hash tables may be necessary for best overall results in some system context. It is also important that the underlying Lisp implementation of hash tables be efficient.

The observed speed-up in Macsyma/Maxima using Allegro Common Lisp version 7 and using these techniques can be shown to be orders of magnitude for large expressions: examples summing up 100,000 terms which are essentially infeasible using the old representation is easily computed using hash coding.

Using a general form of a hash coded sum is not, in our view, a complete substitute for special canonical forms (like those in Reduce, or Macsyma's canonical rational expressions, Poisson series, Taylor series, etc.), where operations like polynomial division benefit from a more explicit representation of mathematical structure such as a sequence of terms by degree. Nevertheless, this form, which is similar to that used by Maple, may be a good compromise for speed without forcing data into a canonical form, and, as long as you are using it, can be made enormously faster than just using lists.

There appears to be no real problem in interfacing these hash table based representations to the rest of Macsyma: the need to convert the hash table forms to lists can be isolated to just a few critical junctures. A single flag is tested in a few routines which determines whether the hash table form should be converted back to the traditional Maxima form. A more sweeping transformation in the system would be to allow hash table representations to be another "first-class" additional internal form, along with rational canonical form, taylor-series form, etc. This hash form would then interact fully with the system, allowing sequences of transformations (say) including pattern matching, human-computer interaction, and commands such as differentiation entirely in the hash table form. It is generally a bad idea to store very large expressions redundantly. We have programmed, as an example, hash table symbolic differentiation.

As is often the case in programming, there is likely to be a trade-off between speed and simplicity in the design and implementation of advanced features in computer algebra systems. If speed counts, it is often possible to get it with more programming.

We have tried to run the Maxima test suite on sourceforge through our modified version of the system, with each hash table returned to list form. Timing on typical "small" problems do not benefit much from our speed improvements. To take full advantage of the hash representation in the test suite, more of the command programs should be altered to operate on hash tables.

The source code for this program is available as one ANSI Common Lisp file "newsimp.lisp" from the author.

6   Acknowledgments

References

[1] B. W. Char, K. O. Geddes, G. H. Gonnet, M. B. Monagan, and S. M. Watt. *Maple Users Guide,* Watcom Pub. Ltd., 5th ed., Waterloo, Ontario, Canada, 1988. Or visit http://maplesoft.com/books/ for a current list of books.

[2] R. Fateman. MACSYMA's General Simplifier: Philosophy and Operation. In V. E. Lewis, ed. *Proc. of the 1979 MACSYMA Users' Conference* (MUC-79), Washington, D.C., June 20-22, 1979, MIT Lab. for Computer Science, 563-582.

[3] R. Fateman and C. Ponder. "Speed and Data Structures in Computer Algebra Systems," *ACM SIGSAM Bulletin, 23* no. 2 (April, 1989) 8–11.

[4] J. Foderaro and R. Fateman. Characterization of VAX Macsyma. In P. S. Wang, (ed.) *Proc. of 1981 ACM Symp. on Symbolic and Algebraic Comp.* Snowbird, Utah, Aug, 1981 14–19.

[5] G. Gonnet. New Results for Random Determination of Equivalence of Expressions. In B. W. Char, (ed.) *Proc. of 1986 ACM Symp. on Symbolic and Algebraic Comp.* Waterloo, Ontario, July, 1986, 127–131.

[6] E. Goto and Y. Kanada. Hashing Lemmas on Time Complexities with Applications to Formula Manipulation. In R. D. Jenks, (ed.) *Proc. of 1976 ACM Symp. on Symbolic and Algebraic Comp.* Yorktown Heights, N. Y., August, 1976. 154–158.

[7] Macsyma Reference Manual, Symbolics Inc. 1988.

[8] Donald Michie. "Memo" functions and Machine Learning, *Nature 218*, April 6, 1968, 19–22.

[9] P. Norvig. *Paradigms of Artificial Intelligence Programming*, Morgan Kaufman, 1992.

[10] C. Ponder. Augmenting Expensive Functions in Macsyma with Lookup Tables. Chapter 9 in *Evaluation of "Performance Enhancements" in Algebraic Manipulation Systems*, Ph.D. diss. Univ. Calif. at Berkeley, Dept. of EECS, also UCB/CSD 88/453 p. 85-100., 1988.

[11] Barry Simon. Four Computer Mathematical Environments, *Notices AMS 37* no. 7. Sept. 90. 861–868.

[12] G. L. Steele, Jr. *Common Lisp: The Language.* Digital Press, 1984.