Haddock's eyes and computer algebra systems Some essays

Richard Fateman
Computer Science
University of California, Berkeley

August 1, 2005

1 Introduction

Symbolic mathematical computing systems (sometimes called Computer Algebra Systems or CAS) are complicated beyond the more common "numerical" computing systems for at least the following reasons.

1. CAS are often expected to deal with issues inherited from (all) the traditions of mathematics. One consequence is that the designers of CAS must accomodate to sometimes-contradictory requirements. *Different* traditions, as well as a general helping of informality, are part of the context in sub-fields of mathematical study. In some cases, explaining what a CAS is doing—even to a reader who is generally savvy about some mathematics—requires introducing formality where it did not exist before. That is, in order to navigate through the possible interpretations of a notation, we may need to remind the reader (or in some cases introduce for the first time) some additional symbolic logic. It is tempting to some writers to introduce a great deal of formalism. In this paper we will try to minimize the novelty for most readers, to some logic and a smattering of λ-(lambda) calculus.

2. CAS must deal with issues that emerge from symbolic computing viewed as a discipline within *computer science*. Since many potential CAS users do not have much formal computer science training, nor do they

think they need such training. That is, in their minds it may seem an unnecessary diversion.

Readers of this paper, including scientists or engineers, and for that matter, professional computer programmers, have enough computing knowledge to write programs—even very long ones and sophisticated ones. Yet such achievements do not necessarily require thinking about certain topics in data representation that crop up within a CAS. The principal issue is one of the meaning of uttering a name. Does one mean *the value associated with that name*, or *the location in which that value is stored*, or something else? It is natural that if you have experience with a traditional programming language such as Fortran or C, you may not have confronted this issue. Even a background in symbolic logic does not guarantee an entirely free ride on such issues. A short course in "compilers" guarantees an acquaintance with the issue.

To try to put a light-hearted spin on this, we will call this the *Haddock's Eyes* problem, in tribute to Lewis Caroll, in a passage we quote later on.

3. For anyone familiar with the tradition of scientific computing, features of *symbolic* mathematical data present far more complications than the more well-known *numerical* data. In particular, the size of a numerical floating-point value is typically fixed for all time at (say) 4 or 8 bytes. Based on this fact it is easy to allocate arrays of such elements, and the arrays too have predictable and fixed sizes. Yet for symbolic data, a value can vary enormously in size over time, as for example the degree of a polynomial can grow or shrink. A striking instance of the difference between these modes is the observation that the determinant of an $n \times n$ matrix of floating-point numbers is simply a number. If the elements are distinct symbols $a_{1,1} \cdots a_{n,n}$ then when written out, its determinant has $n!$ terms. Thus the nature of efficient algorithms and data structures for symbolic computation has a quite different flavor from algorithms for numerical data.

4. The natural datum for many CAS is "a symbolic mathematical expression" in some notation. Most CAS include within that class of expressions, "programs" in one or more familiar languages. That is, since one can manipulation symbolic expressions, one can also construct or modify a program based on some mathematical or algorithmic recipe. This

is related to a generally simpler but more familiar concept: a program that creates programs from text: these are called compilers.

A typical compiler for Fortran or for "C" starts with a data text file constituting a program, and processes it, resulting in a data file constituting an equivalent program, but written in a bit pattern more closely tied to machine language for a particular machine. A CAS can be set up to provide *program-writing programs* that require only the highest-level explanation of what to do, and might, for instance, generate pages and pages of text in the C language. The use of such a system can be nearly miraculous, if all you need to do is (say) provide a formula or two and step back. On the other hand, if the complete recipe does not exists and you have to do "meta" programming to produce a program, the technology can be daunting.

2 What is all this about Lambda?

It is our intention in this section to convey to you a few useful concepts from λ (lambda) calculus, specifically the ideas of *bound* and *free* variables. You may skip this section, and still use a CAS. You might not need to know about these concepts, especially if you don't define functions, or use any variable name twice. But then we hope you won't mind reading this section if you are directed here, later. That might happen if you encounter a "bug", and we point out that the problem is your usage: you tried to oversimplify reality, and must now face up to the fact that you must understand bound variables.

In some sense this comes about because of the common use, if not the inevitability, of ambiguity (i.e. a form meaning with several different interpretations) in informal notation.

Consider the statement "$k^2 \geq 0$." This assertion about some item k is clearly different from "$n^2 \geq 0$" which is a statement about n. This is because k and n are presumably unrelated and could be anything. In these statements they are *free variables*.

Now consider the statements "for every real number k, $k^2 \geq 0$" and "for every real number n, $n^2 \geq 0$".

These two statements *are equivalent* because in the first statement k is *bound* to "every real number" and then the subsequent use of k is not free, but refers to that *binding*. In the second statement, n is *bound* to "every real

number" and then the subsequent use of n falls into the same pattern. In fact we can use any name.

Programming languages use binding. Consider the (Macsyma/Maxima CAS) program fragments `f: lambda([x],x^2+1); g: lambda([z],z^2+1);` . These assign to the names `f, g` a function of one argument, say u which returns the value of $u^2 + 1$. In one case the "dummy name" for the argument is `x`, and in the second, `z`, but the two functions are equivalent, in a λ-calculus sense. If you try `f(3)` you will get 10.

CAS have a few popular features in which binding occurs, but the keyword lambda does not appear, in order to not scare people with an unfamiliar (Greek!) letter. The obvious use for lambda is function definition, and in fact we would ordinarily type `f(x):=x^2+1;` without a lambda. But it is there implicitly.

Where else do we have binding occurring? In most CAS, these are usually restricted to summation, integration, and plotting. That is `k:45; sum(k^2+1,k,0,2);` is usually understood to not involve squaring 45. operationally we define a new function `s(k):=k^2+1` and then compute `s(0)+s(1)+s(2)`. There is no need to use up a perfectly good name like `s`; we can use lambda expressions without names by applying them like functions. For example, `lambda([x],x^2+1) (2)` returns 5. It might be formally "more proper" to have some syntax in Maxima like this: `sum( lambda([k],k^2+1),0,2)` but this was deemed unconventional, and the simpler form prevailed (as in every other CAS).

Why bother? Consider this. `z:k^2+1; sum(z,k,0,2);`
Are we supposed to sum `s(k):=z`, in which case we get `s(0)+s(1)+s(2)` or $3z$? Or are we supposed to identify the k in the value for `z` outside the `sum` with the bound k inside the `sum`? Do we "evaluate" the first argument to the `sum` operation, here `z`, before summing? In which case we would have problems with `sum( if(k>1) then 0 else 1, k, 0, 2)`, because we cannot evaluate $k > 1$ until k has a value.

The same issues crop up with other programs, as mentioned above. he variable of integration is implicitly bound where it appears in the integrand. Also, plotting (perhaps in several dimensions) also binds variable(s).

3 Haddock's Eyes

In this passage from *Through the Looking Glass* by Lewis Caroll, the White Knight proposes to comfort Alice by singing her a song:

> "Is it very long?" Alice asked, for she had heard a good deal of poetry that day.
>
> "It's long," said the Knight, "but it's very, very beautiful. Everybody that hears me sing it–either it brings the tears into their eyes, or else–"
>
> "Or else what?" said Alice, for the Knight had made a sudden pause.
>
> "Or else it doesn't, you know. The name of the song is called 'Haddock's Eyes'."
>
> "Oh, that's the name of the song, is it?" Alice said, trying to feel interested.
>
> "No, you don't understand," the Knight said, looking a little vexed. "That's what the name is called. The name really is 'The Aged Aged Man'."
>
> "Then I ought to have said 'That's what the song is called?'" Alice corrected herself.
>
> "No, you oughtn't: that's quite another thing! The song is called 'Ways and Means': but that's only what it's called, you know!"
>
> "Well, what is the song, then?" said Alice, who was by this time completely bewildered.
>
> "I was coming to that," the Knight said. "The song really is 'A-sitting on a Gate': and the tune's my own invention."

If you visit `http://www.alice-in-wonderland.net/?school/alice1020.html` you can see additional commentary beginning with this...

> Now that is formal logic served up with an apple in its mouth! Those familiar with programming computers in higher-level languages will see there a clear delineation of the difference between a datum, the symbolic name of that datum, the address at which the datum is stored, and the symbolic name of that address. ...

> The whole thing looks to the casual reader a total nonsense, but it's not: the song is A-sitting on a Gate; the song is called 'Ways and Means', but what it is called is not what its name is (as, for example, the once-popular song named "In Other Words" is usually called "Fly Me to the Moon"); the proper name of the song is 'The Aged Aged Man'; but—and here we go beyond commonplace usage but not at all beyond logic—the name of the song has its own nickname, 'Haddock's Eyes'. Confused? Carroll wasn't.

4 Ambiguity, Tradition and Mathematics

In this section we discuss complications caused by conventional mathematics notation, which at various critical points from elementary to advanced, may be ambiguous. That is, the same exact symbolism means two or more different things from different perspectives. Occasionally we encounter so-called "abuse of notation" where a single notation is intended to mean different things to the same person. Sometimes the "abuse" is announced in a technical paper. Sometimes it isn't[1].

Because human users of mathematics are presumed to be familiar with the context in a journal article, the ambiguity can, most often, be resolved. However, if we break up math into small pieces, independent of context, and feed these pieces into a computer system, the CAS will not have information vital to successfully understanding the utterances.

Consider $x = \sin(x)$. This is clearly a false statement if it means "for all x, $x = \sin(x)$." But it might also mean "solve this equation" in which case it is probably equivalent to "$x = 0$" given the usual interpretations of the symbols involved.

Consider $x + x = 2x$. Again this is true for the most common meanings of the symbols. But it is not true if "+" means concatenation. In which case perhaps $x + x = xx$. It is also incorrect to simplify this way if x is an interval, or is infinite or undefined.

[1] We invite a careful literal reader to nitpick this very section to identify places where we ourselves abuse notation by using *an invisible operation* for multiplication: sometimes adjacency means multiplication.

What about "assignments" though? If x has been given the value 43, say by assigning it using the ":=" operation we have borrowed from some programming language notation: $x := 21 + 22$, then the equation $x = 43$ is arguably true, *if we mean "whenever I say x" find the value assigned to x.*

But what if $x := y+3$ and also $y := 40$. Which of the following statements is true: $x = y + 3$, $x = 40 + 3$, $x = 43$? And what if we say $x := x + 1$. Can anything be said about x?

A symbolic system must deal with symbols whose values refer to other symbols (etc.) Just like Haddock's Eyes, referring to ultimately A-sitting on a Gate, which presumably itself refers to a poem.

Mathematics allows us to talk about symbols, but in many application contexts useful results are obtained in ways that do not conform to simplest rules about evaluation or simplification. Yet CAS are usually programmed with just one definite deterministic set of rules for such evaluation.[2]

Different CAS have different rules, although some may be encoded, as in Maxima, inside programs, and not easily displayed as explicit rules. Each CAS has made an attempt to do "what comes naturally most of the time in mathematics". A moderately experienced user of any of the common systems is likely to see examples in which the result is unexpected. A careful reading of the CAS manual may clarify the situation and explain where the fault is (user or system); but, after all, reading a manual at all, much less carefully, is not what most people do!

5 Symbolic variables

I hope you are still with us. We can initially avoid some confusion by separating out two categories of symbols. There are program variables used for denoting quantities. The quantities may change, as the result of running programs or procedure[3] There are indeterminates. These are symbols that

[2]In some "theorem proving" contexts, a CAS might attempt a proof of a statement X in which it contradicts some X in a particular way and then show that leads to a contradiction. Searching for the right route to such a proof some heuristic or randomized method could be possible.

[3]Or "functions". Programmers are often taught that a certain kind of procedure is a function or `FUNCTION` subprogram in which a set of values is given to a procedure, and after some computation, a single value is returned. Thus `f(x):=sin(3*x+1)` might define a function subprogram. There is a relationship between this notation and the mathematical notion of function, but it would have been better for CAS if the term were

do not usually have values. They are sometimes stand-ins for values to be known later, but sometimes they are never known. In this sense ax^2+bx+c is a quadratic in the indeterminate x whose solution can be expressed in terms of indeterminates a, b, and c. It might not be a problem to later convert a to a program variable and then assign $a := 3$, so what's the big deal? Here's the problem. What if we assigned $a := 0$? Then the quadratic formula would not work. What if we assigned $a := 3x$? Then we would have a cubic equation!

A typical trick played on CAS by challengers is to solve a problem and then try to apply the solution beyond its domain of validity. A good CAS will either catch the trickster, or plow on and get some answer anyway. In some cases the answer might even be correct. On a bad day the answer might be a naive blunder.

6 What about data structures?

We are used to a set of notations that are quite non-uniform. Because we are used to it we hardly notice. Because computer programs thrive on uniformity, we can simplify the lives of programmers by modifying the conventional notation. Here's how. The form $\cos x$ becomes `cos(x)`. The form a^b becomes `power(a,b)`. The form $a + b$ becomes `plus(a,b)` with $a + b + c$ becoming `plus(a,b,c)`. The product of a and b, conventionally ab or $a \times b$ or $a \cdot b$ or $a * b$ is `times(a,b)` with the obvious extension for products of more terms. $a^2 + b\cos(x)$ is `plus(power(a,2), times(b, cos(x)))`.

Given this notation (in Mathematica it is called FullForm), writing programs is much simpler. If you want to compute the derivative of an expression S, an operation which requires that you know the main operator (plus, times, cos, ...), the main operator is immediately available as the "Head" of S.

In Lisp, the representation `f(a,b,c)` is written `(f a b c)`. Some people object to Lisp because it has too many parentheses, but as you can see, it has exactly the same number. Lisp also allows you to use odd symbols rather more freely than most other languages, and so we can just as easily use `(+ x y)` in Lisp for `(plus x y)`. There are other advantages to Lisp, including the fact that Maxima is written in it, it is a naturally interactive language in

not so overloaded in the worlds of programming and math. Q. for the reader: in what sense can one "Solve for roots of a function." "Plot a function." "Find the indefinite integral of a function." "Compile a function."

which prototype programs are easily constructed and tested, and there are any number of free implementations for you to use in experiments.

Exercise for the reader: generalize from our comment above and write $a^2 + b\cos(x)$ in Lisp. Lisp uses `*` for `times` and `expt` for `power`.

Everything we want to do to transform expressions to expressions can be done with this representation. $2x + 3x \rightarrow 5x$ can be accomplished by taking a= `(* 2 x)` and b= a= `(* 3 x)`, putting them together in a list `(+ (* 2 x) (* 3 x))` and "simplifying" to `(* 5 x)`. Putting them together in lisp is easy: Let c= `(list '+ a b)`. Simplifying the result is also easy in this case, but in more general circumstances is difficult, or even in some settings, provably impossible.

Sometimes when the easy cases are done in simple-minded ways, they don't generalize very well, or they become very slow for longer examples. Sometimes it is not clear whether an expression is in fact being simplified or made more complicated by a transformation, and the simplification program needs some external nudges. (Consider the equivalent $\sin 2x$ and $2sinxcosx$. Which do you prefer? Numerically evaluating the first one would be faster. Integrating the second one (it is of the form $2u\ du$ with $u = \sin x$) is more obvious.

Even as simple a question as simplifying `(+ (* 2 (+ a b))(* 3 (+ b a)))` raises questions. Take a minute to write that down in conventional "infix" form. Should the answer be left alone, or `(* 5 (+ a b))` or `(* 5 (+ b a))` or even `(+ (* 5 a)(* 5 b))`?

One typical requirement is that the terms in a sum or product be sorted. The sort order must allow us to collect terms. Another requirement is that sub-expressions which are constants, like `(+ 2 3)` should appear as constants. We also generally require that answer that can be written exactly (say, as integers) should be computed exactly. Requiring that `(expt 2 100)` be simplified to 1267650600228229401496703205376 means that writing a simplifier in an number of conventional languages requires a substantial effort. Another point in Lisp's favor is that ANSI Common Lisp supports exact integer and rational arithmetic.

Maxima has a rather elaborate simplifier, including a kind of data-directed aspect: For each operator like cos, sum, *, etc. there is a program attached to that operator (e.g. simp-cos) implementing the simplification below that operator.

A major issue appears only if this technology is pushed to large sizes of complicated expressions. The issue is one of efficiency in re-simplification.

Sub-expressions will be repeatedly simplified to no effect, solely because the program does not know they are already simplified. A good program can be made exponentially more efficient by marking "already simplifed" sub-expressions, and avoiding delving into them again. Maple handles this by putting simplifed expressions in a hash-table. Mathematica handles this by putting time-stamps on sub-expressions. Maxima marks such expressions by changing the operator in `(sin x)` to `((sin simp) x)`.

(If we want to be really accurate with regard to Maxima, the representation of the unsimplified version is `(($sin) $x)` vs `(($sin simp) $x)`. The dollar signs are remnants of a simple-minded attempt to distinguish Maxima variables from others.)

Is there a better solution to all this? In fact, for many many purposes, and for many useful classes of expressions, there is: find a "canonical form" for the class of expressions. In the case of a polynomial in one variable, say x, over the integers, one can encode the polynomial as a list of coefficients. That is, $5x^3+3x+47$ could be encoded as a list, `(5 0 3 47)`. In a representation like this, combining terms, multiplying, adding, sorting, etc. is algorithmically straightforward, and the results are canonical – that is, unique. For more than one variable, say x and y, let us first elaborate on our representation to include the variable: `(x 5 0 3 47)`, and then instead of having coefficients which are variables, allow them to be polynomials in (other) variables. Thus $(y^2+5)x^3+3yx+47$ could be encoded as a list, `(x (y 1 0 5) 0 (y 1 0) 3 47)`.

A variant of this form, closer to what is used in Maxima, assumes that many coefficients are zero. Then instead of storing *all* the coefficients we store only the exponent-coefficient pairs. That is, $5x^100+11x$ is `(x 100 5 1 11)`. The same trick can be used for sparse multivariate polynomials as well.

A few other details must be taken care of. We must designate the ordering of the variables: If there are two, which of x or y should be the main variable? And if there are more, how to order them as well. We also need to be able to use variables with names like $\sin z$ or $\log(w^2+1)$. For this purpose, Maxima associates a list of variables with each expression. In this form, the only thing we need to do to keep expressions simplified is make sure that the leading coefficient is not zero. We also must decide if we are to represent the polynomial 0 by an empty list, or the coefficient 0. Maxima uses 0. A natural extension of this to rational functions, ratios of polynomials, is to have pairs of polynomials in lowest terms. This CRE (canonical rational expression)

representation is discussed in more detail (where?)

Representation is not the only issue. Convenience of programming and speed of execution of algorithms are important as well. Thus we can find ways of multiplying polynomials of n and m terms using nm coefficient multiplications and a sorting operation that in principle could be more expensive $O(nm \log(n+m))$. Is this as fast as can be done? Well if the two polynomials in question are nearly dense, then their degrees will be about n and m respectively. For the dense case, and for large polynomials, it is possible to replace the nm multiplies by $O(k \log k)$ where k is $\max(n, m)$. This FFT algorithm is practical only for rather large degree, dense inputs, and requires a delicate programming touch to do right, and so is rarely used.

– talk about GCD here–

also partial fractions

maybe say exactly what Maxima CRE form is. Why it is so much faster..

The next step: what about hash tables?

Just take the terms and stick them in a hash table. For a term like $345x^2y^{10}z^9$ we use as a key the list [2,10,9] and the value is 345. Advantages

1. We do not have to do a kind of list insertion or list search to add a term into the hash table, so that so operations are probably $O(1)$ rather than $O(n)$ where a polynomial is of size n.

2. Tbe addition of two sparse polynomials represented as HT, if the larger one can be modified, is (probabilistically speaking) $O(n)$ where n is the size of the smaller: there is no sorting.

Disadvantages:

1. The overhead of HT for small expressions.

2. The possibility that HT growth will be needed.

3. The cost of certain operations may be higher. In particular, finding the coefficient of the leading term in a polynomial is fairly direct in CRE form, but is not obvious in the HT representation. Keeping track of it is possible but requires additional work. This operation is specifically required for computing a quotient and remainder.

The speed advantage of unordered representation can be considerable. It is used in Maple, and has been explored as an add-in to Maxima, where on some examples it can be shown to be far faster than the general simplifier.

There are other representation, such as for truncated Laurent series and Poisson series, and modifications of CRE for algebraic number extensions.

6.1 More representations

Maxima does not provide an especially clever way for adding extensions to the repertoire of efficient data structures. While many data structures can be built out of lists and 'unevaluated" functions, these are not very efficient in space or (probably) in time, and so a "professional" version may need some Lisp programming. A system which attempts to overcome the need for the expert hand in adding to the data representation and algorithms is Axiom (formerly Scratchpad), a rival CAS to Maxima, especially since it too is open-source. Axiom, it is claimed, is especially attractive to mathematicians who wish to build upon existing algebraic notations and properties to compose a new algebra of some kind. The MuPad system, as well as the Maple subsystems called Gauss are also efforts to smooth the extension of a CAS. A Maxima system by Bruce Miller, intended to be a hierarchical expression formatter, has some of the aspects of these extensions.

This should not be confused with the *ad hoc* manufacturing of a novel set of behaviors via pattern matching rules. The rule-based approach can work for simple alternatives in an existing system, but especially in the hands of an eager amateur, is prone to being very slow, and because it is incomplete, prone to errors. Few programmers have a positive experience building large systems composed of interacting rules.