Building Algebra Systems by Overloading Lisp

Richard Fateman
Computer Science
University of California
Berkeley, CA, USA

September 19, 2006

Abstract

Some of the earliest computer algebra systems (CAS) looked like overloaded languages of the same era. FORMAC, PL/I FORMAC, Formula Algol, and others, each took advantage of a pre-existing language base and expanded the notion of a *numeric value* to include *mathematical expressions*. Much more recently, perhaps encouraged by the growth in popularity of C++, we have seen a renewal of the use of overloading to implement a CAS.

This paper makes three points. 1. It is easy to do overloading in Common Lisp, and show how to do it in detail. 2. Overloading *per se* provides an easy solution to some simple programming problems. We show how it can be used for a "demonstration" CAS. Other simple and plausible overloadings interact nicely with this basic system. 3. Not all goes so smoothly: we can view overloading as a case study and perhaps an object lesson since it fails to solve a number of fairly-well articulated and difficult design issues in CAS for which other approaches are preferable.

1 Introduction

Given a programming language intended to manipulate primarily numbers, arrays of numbers, and other data objects such as strings, we can consider extending the language to include a new set of objects—let us initially consider them "uninterpreted symbols"—e.g. x, y, z. We then extend to language to allow using these symbols and "algebraic expressions" involving them in as many contexts as possible. In particular these expressions can often be used in place of numbers. Just as we can routinely add $2+2$ to get 4, the extended language should be able to add $x+x$ to get $2x$. The nice analogy may break down in any of several directions. For example, if you wish to perform a loop exactly s^2+3 times, then s^2 must actually be an integer. Another hazard appears in comparisons when the program must resolve whether $(x-y)(x+y)$ is equal to x^2-y^2, when neither is, in fact, a number. Does equality mean "identical in form" or "mathematically equivalent"? A numerical program which, incidental to solving a linear system looks for a "non-zero" pivot (or a "large" pivot) may not be simply run, unmodified, on symbolic expressions.

In addition to such serious issues, there are a plethora of minor issues, which must be balanced against the considerable benefits that accrue simply by allowing much of the design and implementation of a mature language to be inherited. When the symbolic extension idea has nothing special to contribute, the default traditional programming language semantics take advantage of compiler optimizations, etc. Not only (simple fixed-length) arithmetic but other aspects of the host language: procedures, scope, linkage to subroutine libraries, array indexing, loops, error handling are still present.

This unchanged base is a key advantage to designers facing the burden of a CAS design. It relieves the CAS designer of many decisions by providing an immediately available default behavior in the host language. These behaviors may have benefited from an initial design by experienced language experts, and in some

cases may have evolved substantially through informed criticisms by implementers and users, as well as the occasional standardization committees.

The more elaborate CAS view of (say) addition and multiplication must "shadow" the standard arithmetic, *but only when it comes to manipulation of mathematical symbols and expressions.*

The overloading mechanism does not determine the meaning of operations solely in the symbolic expression domain, it only makes it possible to append this to the previous domain. Some of this is relatively arbitrary; this point is made nicely in the introductory computing text by Abelson and Sussman [1] that data-directed abstraction solves interaction between relatively independent modules.

Other design decisions must be made when operators have "one foot in each camp." Returning again to the text by Abelson and Sussman [1] we can find a system that is similar in spirit to what we will discuss here: it uses generic arithmetic with tagged objects to build an arithmetic system in Scheme/Lisp, starting with integers and rationals, to include complex numbers (in two forms), and eventually a polynomial algebra system. This text also explains the failure of a pure "tower" data structure to accomodate the abstractions needed, a point revisited below.

There are several downsides to trying to graft a CAS to a primarily numerical language. Most such programming languages are insufficiently dynamic to, say, harbor the notion of a mutable run-time collection of variables. Consider how you might approach, in C or Java the construction at runtime of a *new symbol variable* that didn't appear in the program text. Could you set it to a value, and then use it as an array index? Or in one of these languages how could you construct a subroutine whose body, say $\sin(1/(x+1))$ is itself expression which is the result of a symbolic computation?

While some of these issues can be overcome to some extent by using a more flexible base language (like Common Lisp), there are still mismatches to a more advanced notion of what a CAS should handle. In particular, a serious concern is that the rich relationships among mathematical algebraic constructs (groups, rings, fields, polynomials, etc) simply cannot be encoded in any plausible way in the usual base language. These difficulties suggest that we should have a design for a language explicitly representing such issues, a language such as Axiom [9]. This criticism may strike you as being unredeemably vague, so we will propose a few illustrations—certainly on the simple side, but perhaps lending some clarity to this issue, before going further with our overloading explanations.

Let us make up a class we will call PCR[x], of polynomials in the variable x over the complex rationals, where each instance is represented as a vectors of coefficients of pairs of rational numbers (real and complex part) and where a rational number itself is a pair of integers (numerator and denominator), and each integer is potentially of arbitrary length. How does z, the zero polynomial in PCR[x], which is arguably a vector of no elements, relate to the complex number $0 + 0i$ the rational number $0/1$ or the integer 0? Are they, for example, supposed to be the same *equal* data structure, even though they are all different types and shapes? One is a 32-bit integer, another is a pair of integers, one is an empty vector, etc. Another example: When you add $a + bi$ to $1/2$ do you get $(2a + 1 + 2bi)/2$ or $(2a + 1)/2 + bi$ or something else?

Writing a program without clarifying these decisions is an invitation to chaos. There is a choice in writing such programs of using any programming language at hand, or using a system whose design includes keeping track of such issues; this is the point of certain CAS, most notably Axiom, a CAS implemented in Aldor (which was at least initially implemented in Lisp).

2 A bit of history

We learn from history that we learn nothing from history.
–George Bernard Shaw

Authors of one system based on overloading C++ claim on their web page, " Its design is revolutionary in a sense that contrary to other CAS it does not try to provide extensive algebraic capabilities and a simple programming language but instead accepts a given language (C++) and extends it by a set of algebraic

capabilities." In fact, overloading, at least conceptually, is probably the oldest approach for building a CAS. FORMAC was built as an overloaded FORTRAN, and is probably the oldest computer algebra system (described in a 1966 paper by Tobey [12]). FORMAC was subsequently upgraded to be an overloaded PL/I. Another historically interesting system was Formula Algol (also 1966, see Perlis [11]). For a survey of these and other early systems see Hulzen[8]. It is true that these early host languages did not necessarily have extension mechanisms for syntactically supporting overloading by writing in the same language: new features were added by writing subroutine libraries and changing or entirely rewriting the compiler for the host language. Yet *to the user*, FORMAC looked like FORTRAN (or PL/I) with some extra features. Systems of macro-preprocessing programs for overloading languages (usually FORTRAN) appeared only slightly later. A good example is the 1976 paper by Wyatt [13] which describes a use of the Augment preprocessor for adding extended precision arithmetic.

The new aspect, if there is one, is that overloading has been made easier. C++ (which emerged in period 1983-1985, and was standardized in 1988) has raised the level of attention to overloading, including syntax extension, by incorporating it into what became a popular host language. It is not necessary to rewrite parts of the compiler.

3   Why Lisp

It turns out to be easy (subject to observing a few tricks) to do overloading of arithmetic in ANSI Standard Common Lisp to accomodate "new" kinds of numbers. The usual argument for defining a new style of generic arithmetic (for Lisp or other overloaded languages) is that other programs, past, present and future in that language can directly access these extensions with little fuss or bother. The analogy with numbers does not always work, as we have already explained, but opening up a different access route to programs already written in Lisp is attractive. For example, one can easily write a numerical quadrature program which can be handed a *symbolic* program and can call a Lisp-language, free, open-source implementation of a symbolic integration algorithm. Lisp has a long history of being able to use dynamic loading of libraries even if they were originally written in other languages; this advantage is no longer so striking as more recently designed languages have adopted this technology.

This last argument does not really distinguish Lisp from Java or Python or C++, but it lead us to a certain asymmetry. Overloading "+" in C++ means that you can use `a+b` for `StrangePlus(a,b)`. In Lisp the change is from (`+ a b`) to (`StrangePlus a b`), or even: (`in-package StrangeArithmetic`) (`+ a b`) and thus not so dramatic. But note that a C++ gcd algorithm would be invoked as `gcd(a,b)` or perhaps `gcd(a,b,target)` but Lisp uses prefix for *everything* and so uses (`gcd a b`).

There are more distinguishing characteristics. Some ideas that are easy to implement in Lisp because it is far more dynamic in nature these languages. For example, in Lisp a new program can be constructed out of Lisp data, compiled into machine code on the spot and then immediately called. We use this feature repeatedly and routinely in setting up some of our generic arithmetic programs.

Each of the extension programs is small, but collectively they represent a core which can be extended by a programmer who need not be a specialist. A programmer should need only a short education, a brief introduction, and a few examples.

It is also convenient in some cases that Lisp supports the mathematical abstraction of integer (not integer modulo 2^{32}), as well as exact rational numbers, as well as floats, double-floats, and complex numbers.

3.1   Approaching Overloading Arithmetic in Lisp

In Common Lisp one could (after ignoring the urgent pleas of the evaluator that you are possibly doing grave damage!) just redefine "+". This is not a good idea: it is better to leave the standard function there, but restrict its domain to ordinary numbers. To do this we define a new `package`, say "GenericArithmetic" or `:ga` for short, in which the definition of "+" in the `common-lisp` or `:cl` package is *shadowed*. In the `:ga` package

we can then define methods for "+" of various objects (e.g. mathematical indeterminates or expressions involving them.) After fiddling with various alternative techniques, we found it is most convenient to create them as members of a class of distinct objects, instances of a newly-defined type specific for symbolic math. We call this type `ma`. The operations such as "+" on such objects will default to the `:cl` "+" when the objects are simply Lisp numbers, but we are free to define them as we wish for `ma` or mixtures of `ma` and numbers.

3.2 Lisp Arithmetic is N-ary

A subtlety in the Lisp system is that +, -, *, / all take a variable number of arguments. To build these n-ary versions we decompose them into calls to two-argument generic routines `two-arg-+` etc., and provide default cases if only one or zero arguments are supplied.

Since the set of methods for each of the arithmetic operators follows a similar template except for the name of the operator and operations concerned with the identity under the operation when one or zero arguments are supplied, they can (and are) defined in one macro-definition, used 4 times.

A similar n-ary situation holds for comparison functions, where Lisp requires that, for example, (< 1 2 3) returns true, since the operands are monotone increasing. We set out a situation in which `two-arg-<` is defined.

The code for all this is about 80 lines. Another 20 is used to declare all the names of overloaded functions. But all this does is provide an empty skeleton: all the arithmetic defaults to `:cl`. Loading a detailed package provides the "worker" functions for combining special types and numbers.

Note that we have not said much about the arithmetic, except that it inherits from Common Lisp. In building a CAS we might wish to know that addition of elements of certain domains is commutative.

3.3 How About Symbolic Math?

The fundamental idea that we found vastly simplified our thinking for overloading Lisp for "symbolic mathematical" computation is that we made a distinction between these objects:

- The Lisp symbol `x`. This symbol can be quoted as `'x`, and has some set of properties relating to its use in the Lisp programming language. For example one can set it to a value by `(setf x 3)` or define it as a function by `(defun x(u)(+ u 10))` or place pointers to its symbol-table entry into a list of three symbols `(list 'x 5 'x)`. It also has a print-name and a package name.

- The mathematical symbol x. We distinguish this mathematical object from anything else in the programming system by wrapping it in a Lisp structure, `ma`. We must be explicit when we cross the boundary from ordinary Lisp to `ma`: Initially we make a math object x by `(ma 'x)`. We appear to cross the boundary in the opposite direction, from `ma` to Lisp when we display an object as, say, a character string. Actually, we just provide a printing method for `ma` objects.

This realization was important in that the traditional temptation (rarely resisted) for Lisp programmers is to make mathematical "stuff" out of ordinary Lisp symbols and lists. It is still possible to do so, but our view now is that memory is cheap and computing is fast; to be careful, it is worthwhile to embed every result in a `ma` wrapper for "public consumption."

Given this framework, we proceed to implement the two-argument *symbolic* arithmetic operations piece by piece. Adding one `ma` object to another requires extracting their contents, making a new list headed by + of the contents, and wrapping it as a math object. `(+ (ma 'x)(ma 'y))` is then equivalent to `(ma '(+ x y))`. Programming the addition of a number to a math object requires two methods, depending on the order of the inputs. They work like this: `(+ (ma 'x) 3)` is changed to `(ma '(+ x 3))`. Given two numbers, naturally we just add them the Common-Lisp way, by inheritance of the ordinary method. The `:ga` specifies that, and so that does not require any program.

Continuing the project to cover all arithmetic in Common Lisp, we can implement all single-argument Lisp arithmetic functions by the same macro-expansion operation if we agree that, for the moment, any conventional operation (say, sin) on `ma` objects will merely result in an `ma` object whose "inside" consists of `(sin x)`.

3.4   Reading and Printing math

The programmable nature of the Lisp reading program allows us to attach the `ma` constructor to a special reserved single character, and also make the quote mark unnecessary. We choose %, so that `%(+ x y)` is analogous to the Lisp quote mark in `'(+ x y)` except instead of making a "quoted object" `(+ x y)`, it makes a `ma` or "mathematical object" $x + y$.

In our early version of the symbolic math package, a newly constructed `ma` object `%x` would display like this: `#S(ma :e x :simp nil)`. That is, its printed display would reflect the fact that it was a Common Lisp `ma` structure with two components, the first of which is the "contents" or expression or "e" part. The second part we haven't mentioned before, but it is a "simp" component, bound to a symbol or nil telling whether the "e" part was simplified or not. For fun, we have provided a print method for the type `ma`— a program that reaches into the form, grabs the "e" component, and prints it as a string of characters reflecting the math object in *infix* form. This is done by

`(defmethod print-object ((a ma) st am)(format st " a" (p2i (ma-e a))))`

where `p2i` is a program which tranverse an ordinary prefix lisp expression in infix order, writing a string. If the result for the above calculation is the math form of `(+ x y)`, then it is printed as x+y. We can push this hack just a bit further and output LATEX code instead, and insert it into this paper. Thus we can print it as $x + y$.

Observe now that `(+ 1 %y)` produces $1 + y$, but `(+ x %y)` is an error: x is an unbound variable.
`(+ 'x %y)` is an error: we have no method "+" for objects of classes (`symbol` and `ma`).
This illustrates the important fact that the Lisp symbol x is not a stand-in for the mathematical object x. That is, they are separate worlds unless we deliberately convert one to the other.

After shadowing all the (finite number of) arithmetic operations defined in the Lisp language, we can declare success and go home. We've used only about 100 additional lines of code (plus 20 lines for declaring a package mostly importing from `:ga`). The prefix-to-infix translation is about 70 lines of code.

Actually, if our goal is to merely demonstrate the possibility of generic arithmetic, we should quit at this point. If we want it to be useful, we are not yet done.

A potential user might try a few examples like
`(+ 0 %x)` and get $0 + x$
`(+ %x %x)` and get $x + x$.
These examples seem to us as not really satisfactory, especially if the programs we intend to write have tests to see if a sequence of arithmetic operations has reduced a math value to zero[1]. One direction is to apply a *simplification program for mathematical objects*. We need to decide when to apply this operation: One choice might be to simplify expressions only before printing. This might be adequate for a demo, but for extensive calculation, each construction of a `ma` object from constituents would sit in memory, and perhaps eventually just fill it up.

Also, if there were any tests, say subtracting two symbolic expressions to test for equality, we might miss a calculation whose complicated result is actually equivalent, after simplification, to 0. Recognizing this equivalence might be critical for terminating a loop. For this reason we probably need for the simplification to be done at least when expressions are compared. It might be prudent to also simplify when the programmer requests it. If we decide to simplify every time an operation is performed, we could then replace methods of just constructing trees with more sophisticated versions, say that `(sin (* n pi))` for integer n would be zero. Programming such methods takes us out of the generic programming motif: how should should

---

[1]Note, reducing a number to zero with numeric operations works the same as before.

programs be written? Some people advocate pattern matching, a paradigm supported by Lisp, but not usually implemented by overloading. We also need to know whether that pattern would work, not only for $n = 3$ but also for a situation in which n can be deduced to be "an integer" though *which one* is unspecified. Simplification is discussed below.

3.5 Simplification

We provide a very brief survey of some of the possible strategies for simplification.

- There are many variations on simplification. Some are essentially "demoware": easy to write, cheap, fast, and adequate for simple examples. Some are more difficult to write, sometimes more expensive to run, but generally far more effective. No method is completely effective for "all expressions"[2].

- We haven't defined "effective". One definition would require that if $E(x)$ is 0 for all values of x, then an effective simplifier must reduce $E(x)$ to zero. Except for expressions that are composed of a very limited repertoire of operators, there are no effective zero-equivalence algorithms.

- Nevertheless, there are different methods that do more or less depending on domains and the intentions for further processing. Since one method does not dominate all others, it must be possible to choose amongst them (or perhaps try them all!).

- Even the cheap methods become slow if they are implemented naively.

- To implement even the cheap methods as expressions scale up in size, efforts must be made to mark "already simplified" subexpressions.

There is a substantial literature on the topic of simplification. A *simple-minded simplifier* `simpsimp` we wrote initially for MockMMA and then modified for Tilu (1996), has been further modified slightly to show how such a facility can be integrated into this design. A second (polynomial and rational function simplifier) from the same MockMMA code base provides `ratexpand` and `ratsimp`. Each of these cancels common factors and can determine algorithmically if a polynomial is zero or not. The `ratsimp` program generally keeps factors it encounters explicitly when it can, which is a benefit to some manipulations. The `ratexpand` program provides a canonical form based on a ratio of "expanded" polynomials.

In this situation of multiple simplifiers, we use the "simp" part of the `ma` "wrapper" to mark that the object contained in the `e` field is simplified, and by which simplifer. We could also record some settings of parameters in the simplifier at the time this was done, as well as other information that might be convenient at a later time, including a list of the variables in `e`.

3.6 Evaluation

The idea of evaluation is to associate some indeterminate, say x with a value, and use that value consistently in an expression E instead of x. The value for x could be, for example, the number 3 or the math expression $y - 45$. Evaluating $E = x - y + 45$ with $x = y - 45$ should provide $(y - 45) - y + 45$, or after simplification, 0.

One model for evaluation whose semantics is commendably clear, is that of substitution for a specific variable. This model inserts a copy of the new value in the expression E each place there is an occurrence[3] of x. It might be a good idea to re-simplify. We insist that the new and old "values" must be mathematical objects, as distinct from Lisp symbols. Overloading the conventional Lisp `subst` program is a possibility, but it turns out to have such a plethora of options so that it seemed prudent to call the math substitution program `masubst`. Thus `(masubst %(- y 45) %x %(+ x (- y) 45))` produces a value equivalent to 0.

---

[2]There are "undecidability" theorems to this effect.
[3]Technically, we should point out that this is for free, not bound occurrences.

Another model, whose semantics tends to be murkier is to reconsider the expression E, at all levels, identifying with each and every name in the Lisp system, variables or functions, the corresponding math variables or functions. Some of the murkiness is whether you really want to "evaluate" $\sin \pi$ using Lisp's value for π in which case you get 1.2246063538223773d-16, but is only an approximation, or if you want to *simplify* the same object to the exact value 0. In any case, the function `meval` implements this model.

3.7 Input Parsing

There are many parsers for "infix math" written in Lisp, and almost any of them could be used in conjunction with this system. As a sample, we show how to use a particular parser by attaching a program to a symbol (here we use $) so that the rest of the line following that symbol is parsed as infix mathematics into Lisp. By prefixing that with a %, we convert it to our "math" representation.

For example,

```
: (setf trythis %$sinx^2+45sin^2x+abcosc
)
sin(x^2)+a*b*cos(c)+45*sin(x)^2

: '$q(a,b,c):=-b+sqrt(b^2-4ac)/2a
(defun q (a b c)
  (+ (- b) (/ (sqrt (- (expt b 2) (* (* 4 a) c))) (* 2 a))))
:  $cos pi
-1.0d0
```

The input for this particular parser is closer to spoken or written mathematics than (say) FORTRAN. First developed as part of Tilu, it includes as a subset a traditional infix language, but has enough additional ambiguous grammar rules for dealing with adjacency of symbols that people who leave them out may be pleasantly surprised to see the right thing sent back out to them. Ambiguous input like `a(x+y)` provides two parses, the function application and the product`a*(x+y)`. The user has an opportunity to choose. The user may also choose to permit multi-character identifiers or add new names for "built-in" functions. It is based on Norvig's [10] context-free parser.

4 The Downside of Overloading in Lisp

The nature of arithmetic computation in Lisp is that, for the most part, operators like "+" and "*" need know nothing about their operands until after the operands are evaluated. Only then are their types available for consideration. Therefore if you use Lisp operators always for simple arithmetic on double-precision floats, but keep this knowledge a secret from the Lisp compiler, Lisp will still make arithmetic-type discrimination decisions at runtime repeatedly and wastefully. Lisp supports a method to `declare` types in Lisp, in which case the compiler will produce fairly conventional and fast assembly code. The tradeoff is that such program and will not work correctly when fed (say) arbitrary-precision integers. In our programs, the distinction between (say) a "symbolic math object" and an ordinary Lisp integer must be made at runtime because, in fact, the same operator program must deal with either kind of argument, and ordinarily will not be able to tell which will occur in advance.

By a certain amount of cleverness, some specialization may be applied to eliminate or reduce runtime checking by the object system compiler. A far more thorough approach to such specialization, requiring essentially that we generate code specific to the types of arguments, can be based on our "application based" knowledge. Inserting such code is facilitated by Lisp's macro-expansion capability. By additional analysis at compile time, when the full power of the Lisp system is also available, we can write fully "type-declared" and specialized code.

5 Composing Overloads

We have actually built various other overloads on top of generic arithmetic before building the CAS features; later we came to see the :ma package as a good vehicle for explaining what we have done.

This section interweaves some of those concrete examples especially showing the prospect of using *several types simultaneously.*

This is most likely to provide the positive demonstrations needed to convince the reader of the value of this approach.

In these examples we show that combining special operations *can be* as simple as writing programs that simple compose the ordinary Lisp functions, and execute what appears to be normal code, though in some respects it differs, using novel operations.

Here is a recursive definition of Legendre polynomials of the first kind.

```
(defun p(m x)(cond ((= m 0) 1)
                   ((= m 1) x)
                   (t (*  (/ 1 m) (+ (* (1- (* 2 m)) x (p (1- m) x))
                                     (* (- 1 m) (p (- m 2) x)))))))
```

Whether this function takes symbolic or solely numeric arguments depends on whether `p` is defined in a package which inherits the symbolic arithmetic operations.

We introduce the definition of `p` here so we can use it in the next section.

Automatic Differentiation (AD)

In the model of arithmetic introduced for Automatic Differentiation [5, 6], an ordered pair $< a, b >$, represents the value of a function of x, say $f(x)$ at a particular (but unstated) point p, i.e. $a = f(p)$. The second value in the pair represents the derivative of f, also evaluated at p, i.e. $b = f'(p)$ in the usual, but uncomfortable, notation for this concept. We have written a package to compute with these pairs. The operations are keyed about data objects which we manufacture using a constructor `df`. A more extensive discussion of AD including elaborations to multiple variables, higher-order derivatives, applications and algorithms, is beyond the scope of this short paper. Nevertheless, here's an example of the interaction.

```
: (p 3 1/2)              ;exact value of legendre[3,1/2]
-7/16
: (p 3 0.5)              ;single-float value
-0.4375
: (p 3 (df 0.5 1.0))     ;single-float value, as well as derivative
<-0.4375, 0.375>
: (p 3 (df 0.5d0 1.0))   ;double-float value and derivative
<-0.4375d0, 0.375d0>
: (p 3 (df 1/2 1))       ;exact value and derivative
<-7/16, 3/8>
: (p 3 \%x)              ;exact value in terms of x
(1/3)*((5*x)*((1/2)*((-1)+(3*x)*x))+(-2)*x)
: (simp (p 3 \%x))       ;exact value, somewhat simplified
(1/3)*((5/2)*x*((-1)+3*x^2)+(-2)*x)
: (simp (p 3 (df \%x 1)));exact value and derivative, simplified
<(1/3)*((5/2)*x*((-1)+3*x^2)+(-2)*x),
 (1/3)*((-2)+15*x^2+(5/2)*((-1)+3*x^2))>
: (ratexpand (p 3 (df \%x 1)))
<(1/2)*((-3)*x+5*x^3), (1/2)*((-3)+15*x^2)>
```

```
;same answer, using canonical rational expansion
```

There are a few more kinds of arithmetic overloads that have some popular support and could easily be implemented on top of the `:ga` package. Some are indicated in additional sections below.

Intervals

Real intervals $[a, b]$, represent all numbers x where $a \leq x \leq b$ for real numbers a and b. Should a and b be allowed to be `ma` objects? Probably not unless those `ma` objects also happen to be specific real numbers like π or $\sqrt{2}$ which can be compared. Perhaps unions of real intervals should be allowed, e.g. this is the case in Maple and Mathematica. The literature on "reliable computing" has grown to describe related techniques. This can be combined with AD in particular for computing a Newton-type iteration modified for intervals.

Bigfloats

Arbitrary-precision floats or bigfloats are software-based extensions of the floating-point idea, with fraction and exponent fields represented by variable-length fields. Various implementations have been available for 35 years or more, but it appears that one of the most widely-used is the Gnu GMP package [7]. It, and its extensions probably fit well within the generic framework, although coercions for combination among exact numbers or machine and software floats of different precision must be developed. Within the bigfloat design there are several possible variations other than GMP that could be encoded, including fixed-size fraction and extra-large exponent, or a sum-of-floats representation.

We have interfaced Lisp to the GNU multiprecision package (GMP) for several projects in the past, and the arithmetic itself presents few difficulties, if a functional model is used. A closer fit to GMP, in which operations and storage are intertwined, looks less like generic arithmetic. (Consider, for example, a single "operation" addmul(x,y,z) that destructively replaces the value of x by x+y*z; this is not available except perhaps as a compiler optimization, in Lisp.) The memory management requires some minor design: we need to consider when the *storage for a number which has been allocated by GMP* should be returned. We have implemented this by using a "finalize" feature, which essentially provides advice to the garbage collector. This finalize feature is available in several Lisp implementations although it is not required by the ANSI standard. GMPZ (integer) is probaby of interest in Lisp operations only to the extent that it might be faster, perhaps on very long operands, than Lisp's ordinary bignum arithmetic. Similarly for GMPQ, (rationals). The bigfloat operations in GMPF arithmetic do not have equivalents in the Common Lisp standard, although several bigfloat Lisp libraries exist. GMPF does not include transcendental functions and so cannot provide an overload for sin, cos, etc. There is an extension GMPFR for *rounded* bigfloat arithmetic which includes such operations. Oddly, there appears to be no straight path to convert a GMPZ to a GMPF; our program could do so by converting the integer to a character string and then to a bigfloat. This operation presumably could be coded more efficiently, if necessary.

Extended Rationals

In an unpublished 1994 paper [3], we proposed to use the rational forms 1/0 and 0/0 to extend the rational arithmetic in Common Lisp. We programmed a "projective" model including an un-signed infinity and a "not a number". This is useful for example in allowing the evaluation of $a + b/c$ to a in the case that c is 1/0. It is not necessary to signal an error. We have also programmed an alternative "affine" model which has four extra elements, -1/0, +1/0, 0/0, and a negative zero, (0/-1). (The ordinary 0 is "positive" zero.) This is comparable to those features in the IEEE 754 floating-point standard. We have written elsewhere [3] about the implications of this change, especially for use in interval arithmetic.

Matrices

A version of matrix arithmetic is possible, simulating, in effect, the arithmetic of Matlab or similar programs. The elements of the matrices could be conventional complex-float numbers, or almost anything else that supports (not necessarily commutative) multiplication and addition.

Complex Numbers

Common Lisp already supports complex arithmetic as represented by pairs of rationals and floats. It does not allow extended rationals, intervals, mathematical expressions, etc. Building a new version of complex numbers with such extensions is functionally straightforward, but defining some of the content is a challenge, since we would need to carefully observe the elementary functions and their branch cuts' behavior with respect to (say) infinite imaginary part. An alternative representation in polar coordinates has some attraction as well. We can even create a polar form whose modulus and arg are symbolic expressions from the `:ma` package. resolving combinations of different forms may require reference to the expected type of the eventual storage of the result, though Lisp does not require or even expect that variables have types.

6 Comments on Overloading and Symbolic Math

We have already mentioned the inability of overloading *per se* to resolve questions of how to structure a system for doing "computer algebra". Even in our small example, we illustrated embedding symbolic math in the automatic differentiation (AD) pairs of `df` structures (which works), or the reverse. This latter choice makes almost no sense mathematically, since AD pairs must always be considered with respect to a particular variable. Given this restriction, we didn't program it.

On the other hand, it would be trivial to write a few more methods and *appear to do so*. Nothing in the generic arithmetic framework would prevent it. Even among the programs we wrote, we can ask if the existing methods in AD form, in some sense, a complete coherent whole. Did we really do the right thing for computing a derivative of $|x|$? We cannot tell for sure: the tools of "data structuring", objects, and an inheritance hierarchy are insufficient guidance for structuring mathematical categories and their relationships, and thus overloading a conventional language soon reaches limits.

Some of the more pressing advanced problems in applying CAS to problems in applied mathematics require yet another, perhaps orthogonal kind of reasoning, not about data types or mathematical categories, but about relationships of values. For example dealing with geometric or range assumptions in finding solutions to algebraic or differential equations, domains of integration problems, or parameter spaces in (for example) reasoning about matrices.

Some aspects of these issues may be solvable by solution space decomposition based on systems of polynomial equations. Such techniques fail when log, exp, or other functions are introduced to the mix. These may require something akin to theorem proving for resolution, and are at an even further remove from the overloading technology discussed here.

While we are powerless to prevent programmers from constructing yet another C++ overloaded library to provide CAS capabilities, we hope that these comments will provide some insight in case it is attempted. We are encouraged that some people have actually used others' C++ library (Ginac in particular). Setting out to build a "new" library may take years, and history has shown than much of that time and all of the funding can be occupied by simply repackaging the "easy" parts. Here we have recycled code that was originally written years ago.

7 Conclusion

Generic arithmetic can be easily supported in Common Lisp. We show how it can support one application: symbolic mathematical expressions, and briefly introduce automatic differentiation. These results are not surprising, though we hope that our presentation has some novelties. What is perhaps surprising is that no one has written this paper earlier, since the Lisp language is such a fine host for this effort.

We also illustrate that, as neat as it can be for particular calculations, the simple approach of build a CAS via "libraries plus overloaded language" does not provide a convenient framework for advancing important representation issues: the typical object-oriented data structure design typically allows the construction of nonsensical compositions. While it is possible to avoid the resulting difficulties, the first step is to realize that these problems are ignored at one's peril.

Appendix G: Generic Arithmetic, ga, ma

All the files are available in `http://www.cs.berkeley.edu/~fateman/generic`. The `:ga` code

```
ga.lisp
;; Generic Arithmetic package
;; based on code posted on newsgroups by
;; Ingvar Mattsson <ing...@cathouse.bofh.se> 09 Oct 2003 12:16:09 +0100
;; code extended by Richard Fateman, November, 2005

(eval-when '(compile load) (load "packs"))

(provide "ga" "ga.lisp")
(in-package :ga)

(defun tocl(n) ; get corresponding name in cl-user package
  (find-symbol (symbol-name n) :cl-user))

;; a tool to manufacture two-arg-X arithmetic for X=+,*, etc
(defmacro defarithmetic (op id0 id1) ;id0 = no args, id1 =1 arg
  (let ((two-arg
 (intern (concatenate 'string "two-arg-" (symbol-name op))
 :ga ))
        (cl-op (tocl op)))
    '(progn
       (defun ,op (&rest args)
         (cond ((null args) ,id0 )
               ((null (cdr args)),id1)
               (t (reduce (function ,two-arg)
                          (cdr args)
                          :initial-value (car args)))))
       (defgeneric ,two-arg (arg1 arg2))
       (defmethod ,two-arg ((arg1 number) (arg2 number))
 (,cl-op arg1 arg2))
       (compile ',two-arg)
       (compile ',op)
       ',op)))
```

```
(defarithmetic + 0 (car args))
(defarithmetic - 0 (two-arg-* -1 (car args)))
(defarithmetic * 1 (car args))
(defarithmetic / (error "/ given no args") (car args))
(defarithmetic expt (error "expt given no args") (car args))


;; defcomparison is a tool to manufacture two-arg-X numeric comparisons.
;; CL requires comparison args to be monotonic. That is in lisp, (> 3 2 1) is true.

(defun monotone (op a rest)(or (null rest)
        (and (funcall op a (car rest))
    (monotone op (car rest)(cdr rest)))))

(defmacro defcomparison (op)
  (let ((two-arg (intern (concatenate 'string "two-arg-"
      (symbol-name op))    :ga ))
        (cl-op (tocl op)))
    `(progn
        (defun ,op (&rest args)
         (cond ((null args) (error "~s wanted at least 1 arg"  ',op))
               ((null (cdr args)) t) ;; one arg e.g. (> x) is true
               (t (monotone (function ,two-arg)
    (car args)
     (cdr args)))))
       (defgeneric ,two-arg (arg1 arg2))
       (defmethod ,two-arg ((arg1 number) (arg2 number)) (,cl-op arg1 arg2))
       (compile ',two-arg)
       (compile ',op)
       ',op)))

(defcomparison >)
(defcomparison =)
(defcomparison /=)
(defcomparison <)
(defcomparison <=)
(defcomparison >=)

(define-symbol-macro ga::* cl::*);we want to keep top-level symbols * + /
(define-symbol-macro ga::+ cl::+); for the R-E-P loop.
(define-symbol-macro ga::/ cl::/)

;; these two macros use ga::+, not cl:+, but should compute complex "places" more carefully
(defmacro incf (x &optional (delta 1)) `(setf ,x (+ ,x ,delta)))
(defmacro decf (x &optional (delta 1)) `(setf ,x (- ,x ,delta)))

(defun re-intern(s p)  ;utility to "copy" expressions into package p
  (cond ((consp s)(cons (re-intern (car s) p)
(re-intern (cdr s) p)))
```

```
((symbolp s)(intern (symbol-name s) p))
(t s))) ;nil, number, string, other
```

`ma.lisp`

`rather than insert ma.lisp code here/ visit web site`

The programs for converting prefix to infix expressions (in ma.lisp).

`rather than insert p2i.lisp code here/ visit web site`

Some interfaces and some tests. There is a simplifer (too long and not interesting), in the file `simpsimp.lisp` with a program in it called `simp`.

`insert examples from ma.lisp code`

The file simpsimp is about 350 lines of code. The rational expression (ratio of polynomials) simplifier in several variables, is larger, but much ultimately both faster and more effective. It is about 1000 lines of code.

References

[1] H. Abelson, G. Sussman, *Structure and Interpretation of Computer Programs,* 2nd edition 1996. MIT Press. Full text available on-line. http://mitpress.mit.edu/sicp/full-text/sicp/book/book.html

[2] "A Short Note on Short Differentiation Programs in Lisp, and a Comment on Logarithmic Differentiation," *SIGSAM Bulletin Volume 32, Number 3*, Sept., 1998,pp. 2-7. `www.cs.berkeley.edu/~fateman/papers/deriv.pdf`.

[3] , R. Fateman and Tak Yan, "Computation with the Extended Rational Numbers and an Application to Interval Arithmetic" 1994// `http://www.cs.berkeley.edu/~fateman/papers/extrat.pdf`.

[4] R. Fateman, "Building Algebra Systems by Overloading Lisp: Automatic Differentiation", (submitted for publication).

[5] A. Griewank, "On Automatic Differentiation," in M. Iri & K. Tanabe (Eds.) *MATHEMATICAL PROGRAMMING,* Kluwer Academic Publishers, 1989, pp. 83-107.

[6] A. Griewank and G. Corliss (eds), *Automatic Differentiation of Algorithms*, SIAM, 1991. esp. paper by L. Rall

[7] The GMP web page is `http://www/swox.com/gmp`.

[8] J.A. van Hulzen and J. Calmet, "Computer Algebra Systems" in *Computer Algebra Symbolic and Algebraic Computation*, edited by B. Buchberger, G.E. Collins et al, Srpinger- Verlag, 2nd ed., 1983.

[9] Richard D. Jenks and Robert S. Sutor, AXIOM: *The Scientific Computation System,* Springer-Verlag, 1992.

[10] P. Norvig, *Paradigms of Artificial Intelligence Programming*, Morgan Kaufmann, 1992.

[11] A. Perlis, Itturiaga, R., Standish, T. "A Definition of Formula Algol," in *Proc. Symposium on Symbolic and Algebraic Manipulation of the ACM*, Washington, D.C., March, 1966.

[12] R. G. Tobey, "Experience with FORMAC algorithm design", Comm. ACM 9 no. 8 (1966) p. 589–597.

[13] W. T. Wyatt, Jr. D.W. Lozier, and D.J. Orser, "A Portable Extended Precision Package and Library with Fortran Precompiler" ACM Trans on Math Software ISSN:0098-3500 1976, vol. 2 no. 3 p. 209–231.