Comments on Extending Macsyma with New Data Types

Richard Fateman
Computer Science Division, EECS
University of California, Berkeley

January 16, 2006

Abstract

Any design for a computer algebra system (CAS) naturally includes a set of data layouts for symbolic or mathematical algebraic expressions intended for use by built-in or user-written programs. The CAS cannot build in all plausible data designs, but supports those of most interest to the programmers. In such a situation it is almost inevitable that some new data encoding idea will come to mind and with it an interest in adding additional data forms. The motivation may be for compact representation, or efficient (fast) manipulation, or for other reasons such as interchange with other programs. Most CAS therefore include at least one way to extend the base set of operations. We comment on the kinds of extensions possible, using Macsyma as an example CAS. The particular interest in Macsyma and its open-source sourceforge variant "Maxima" is that a substantial group of very-losely coupled independent researchers are approaching this problem and may benefit from some guidance. Some the observations apply to other CAS, even though they are not open-source.

1 Introduction

When someone proposes to augment a CAS with a new data type, "computational completeness" is not the motivation since any data could in principle be encoded by pre-existing symbolic forms. In particular "algebraic function forms" can simulate structures using existing built-in facilities.

For example here's a "functional" record made of an otherwise undefined function form that might be used in Macsyma: `PersonnelRecord (LastName, FirstName, Birthdate,...)` We can easily define accessors that can decompose a particular instance of a `PersonnelRecord` called `PR` such that `GetLastName(PR) := part(PR,1)`. With a two-line change to Macsyma[1] and a few added programs we can make this look "object oriented". Because the operator "." is already used for matrix multiply, we use "@" to separate an instance name from a field name: `PersRec@LastName` accesses a field defined in the class of `PersonnelRecords`. We need to also be able to assign values, as in `PersRec@LastName:"Smith"`. The other kind of fundamental built-in tool for extension to data types is arrays: indexed objects. Together, these mechanisms suffice for an encoding of different parts of mathematics, as well as other kinds of data processing. Computational completeness does not, however, mean efficient.

The Macsyma system originally tried to address efficiency plus generality by providing a framework for different data structures. Several variations were included in the initial design. Given the eclectic nature of the design, it remains plausible to add additional features, not necessarily written in Lisp, without doing violence to the original design. It has been our attitude, especially in the realm of free / open source software, that if someone exhibits a program P that is a faster way of computing something of interest (say, factoring a polynomial) than the routine in Macsyma, the right response is to call P from Macsyma.

[1] the definition of `mset`

There are other CAS, more recently designed, where the intention is to provide extension tools in a mathematical abstraction setting. Axiom and MuPAD each have methods for users or programmers to incorporate new domains and describe their salient characteristics so that existing programs can generalize over those domains. There appear to be at least two problems here. One is that the designers must be prescient about the nature of future extensions for those extensions to fit well with the system. Additions that do not fit the pattern of expectations will not be simple to assimilate. The second problem is that, except for a handful of experts whose skills must include knowledge of esoteric systems issues and mathematics, the extension facility are too difficult to use. Other systems, less well known, include Gauss (within Maple), Newspeak (designed by John Foderaro at UC Berkeley, 1984), Weyl (Richard Zippel).

While providing a mathematical framework for building additional algebraic structures on top of a foundation is undeniably an appropriate esthetic direction, this does not replace the need for a pragmatic framework for extending systems.

We will discuss how Macsyma (as well as its free version Maxima), written in Common Lisp, provides such a framework, and how this affects the set of facilities: including various simplification and manipulations programs as well as a number of command / workbook front ends.

2 Extensions to functionality and representation

Over the years, a number of facilities have been added to the original (circa 1968) Macsyma design. Their integration into the Macsyma system has not always been as complete as might be useful.

The default representation for the functional form indicated above is a version of an "algebraic tree" representation, handy for some algorithms, especially heuristic simplification. For input and display purposes certain well-known arithmetic operations are defined with infix syntax, but are still functional, in reality. That is, you can type `x+a` or alternatively, `?mplus(x,a)`, and you will see displayed $x + a$. But really under the covers, that value is a Lisp list, something like `(mplus x a)` [2]. Assume you don't want to use the default encoding, but rather some kind of special form using hash tables, arrays, bit-strings, pointers to disk files, or pointers into data produced by other (non-Lisp) programs. You would not be the first to try any of these. In fact the Macsyma system was built with the objective of hosting a variety of (perhaps redundant) representations. The lesson from an earlier system (Mathlab68)[3] was that an alternative "rational representation" was very useful. Its `RATSIMP` command converted to a canonical rational expression (CRE) form and then back out; the name was re-used in Macsyma. The Macsyma system went further to expose for the user this internal mrat (CRE form). CRE form is a more efficient encoding (compared to "trees") for polynomials and ratios of polynomials. This representation provides necessary support for several important algorithms (e.g. factoring, GCD, in addition to ratsimp).

After the initial Macsyma design, additional special representations were incorporated.

1. mrat / TPS (Taylor series). This resembles polynomial form used in mrat representation, but the exponents need not be integers, the algorithms require some attention to the level of truncation, and there are several models for dealing with multiple variables.

2. mpois (Poisson series). This representation is used for trignometric series of a particular kind, frequently used in celestial mechanics calculations.

3. bigfloats. This is a software representation of approximate real numbers. Most computer hardware supports floating-point numbers, and most languages (including Lisp) provide supporting software for floats. However, the extension to arbitrary precision requires some work. Some libraries, typically written in C, have been built; most CAS these days have their own version.

[2] More precisely, the Lisp version is `((mplus simp) $a $x)`. The complications you see in this form are a mixture of subtle efficiencies and historical quirks.

[3] Not related to Matlab

Some of these have variants, e.g. algebraic "rats" are also defined.

Furthermore, some conventional expressions are treated unconventionally. The `simptimes` program not only has special cases for mrat forms, but also makes special recognition of multiplication of two equations and of multiplications of scalars times matrices or scalars times equations. Say we wish to introduce a new kind of scalar number, an interval. In Maple this is called a range; in Mathematica, an Interval. Often those cases that you might have thought were already settled, become open sores. What else might break when it can be shown that $x^2 - 1$ and $(x-1)(x+1)$ evaluate to different values for some particular x? (Choose $x = [-1, 1]$, a real interval. The first evaluates to $[-1, 0]$, but the second to $[-4, 0]$.)

How should a multiplication of a matrix by a real interval be handled? Or by a complex numeric constant? We revisit this issue with our specific example later in this paper.

Recent papers we have written include suggestions and benchmarks for using an alternative bignum package (GMP, the Gnu Multiple Precision package) with Macsyma, and storing polynomials by stringing together coefficients into one long integer. Another paper proposes using hash-tables for storing exponent-coefficient pairs in a sparse polynomial representation, instead of vectors.

Another recent paper suggests using a special representation for rational functions in one variable, using a pole and residue (PR) representation. In this papers we will follow up on how can this be incorporated into a Macsyma system. By demonstrating this in some detail we hope to provide the PR representation and also provide a guideline for others who might wish to interface other programs, written in Lisp or not, to the framework of the Macsyma system. Only as much of the Macsyma system as is useful need be adapted, as can be seen.

3 Can't we just use Object Oriented (etc) Extensions?

It would be handy if a good programming language, one that has good support for extension of its base system, could do what we describe below, automatically. The most prominent of systems promoted to support such activities in CAS is probably Axiom, now an open-source system, written in the Aldor language. While Axiom may smooth some issues, it is our belief that it can get the details right only by a fair amount of explicit instructions. The reasons for this should become apparent as we explore the territory.

4 Talking it through

Adding a representation R to Macsyma can be done at a number of levels of sophistication. As the level increases, more programming work is needed, and the possibilities of introducing bugs increase. We outline the major steps here.

4.1 Level 1

Assume that the starting point of the representation R is so alien to Macsyma that it cannot be parsed or displayed in the Macsyma formalism. It may help to have a concrete example or two in mind. Imagine that you wish to introduce to the system "Sound" which can be spoken into a microphone and played on a speaker. Or perhaps "hand-drawn graphical diagrams". Here is what one can do:

Write a few programs

- `ReadFromUser()`
- `DisplayR(AnRObject) or PlaybackR()` a display program which displays or plays aloud something useful if given something in R format.
- optionally `ReadRFromFile(filename) and WriteRToFile(filename)`

This would not be very interesting without at least one command that operates on R objects. E.g. `Operate(cmd, Robj1, Robj2, ...)`. The "cmd" might be specific to sound (e.g. change pitch), or to drawings (e.g. rotate).

We expect there is at least one point at which the previous work in the CAS and the newly introduced representation "touched". Say "convert the sound into numeric values" or perhaps "read a formula out loud". Without this, there does not seem to be much reason to incorporate the feature in the CAS.

As an actual example from the past, we describe briefly the interface between Macsyma and Matlab, (a numeric matrix-oriented program) that we wrote at Berkeley in the early 1980s. At the time it seemed that a good way to introduce linear algebra into the system was to completely hijack the routines. Matlab had such a flavor though hardly as exotic as sound or graphics. In this case the objects in question were arrays of floating-point numbers. The linkage worked like this. A command is issued from Macsyma to start a process running Matlab, (`matopen()`), and then a sequence of Macsyma communication commands can be issued.

These include:

- `matput(M,A)` put the matrix M into matlab format, and name it A.
- `newA:matget(A)` get the matrix A back from matlab and put it into Macsyma format. Store as `newA`.
- `matrun(command)` run the (string) command in the Matlab system.

That's about all that was needed.

A different kind of interface, written more recently (2004), feeds GMP with numbers originally created in Lisp. The rationale for this package is that (historically) most Lisp implementations were not especially fast for really long number manipulation. Thus it may pay for a CAS (or even Lisp) to have commands like `gmpnum:intoGMP(lispnum)`, `lispnum:outofGMP(gmpnum)`, with a suite of programs such as `AGetsAXplusB (Agmp,Xgmp,Bgmp)` which computes `A :=A*X+B` all as GMP bignums.

There are various options for the treatment of such items by display. One way is for the display program to realize that it has a GMP number as an object to display. It can then call a program to convert that GMP number into a string, and display the string. This solves the problem since all modern Lisps can deal with strings. Another possibility is a program that converts a GMP number to a Lisp bignum, and then uses whatever facilities are available. (Speed on output is usually not an important criterion for CAS.) Another option may be the use of a GMP direct-to-output program, especially useful for numbers that are so horrendously long they may exceed the Lisp bignum limits. This would be difficult to fit into a display which included a GMP number as part of an expression, say $ax + b$ with a, b GMP numbers.

Useful as these adjuncts might be, they are not really participating fully in the CAS. They are more like hangers-on using the front end and some of the display, but mostly not interacting with Macsyma. We can imagine other links at this level, even to other computer algebra systems. Sending a package of computations to (say) Singular, GAP, NTL, MAGMA, Maple or Mathematica.

4.2 Level 2

Suppose that there are two representations that can, at least in principle, be used for the same underlying concept. For example, $1+x$ could be a truncated Taylor series or a CRE polynomial or an ordinary algebraic expression. Let us assume we have A=$1 + x$, a CRE polynomial and B=$1 + x + O(x^2)$, a series. Then the result of multiplying `A*B` could be done by converting `B` to a polynomial and producing $x^2 + 2x + 1$ or by converting `A` to a (truncated) Taylor series, $1 + 2x + O(x^2)$. There is no automatically "correct" result in such cases: the system might be most prudent to require the user to specify a choice. In general the choices might include more than two choices. One can consider combining two operands by elevating each of them to some superclass (perhaps one of many possible) in which they can both be represented and where the operator combining them is meaningful.

If this is to be done automatically it may be the job of the simplifier to take a good guess at such coercions. In Macsyma this is not done in an elegant way, but by augmenting the simplifier programs themselves. Typically one must change programs internal to Macsyma to be cognizant of the new data types, and enforce the desired inheritances and coercions. An automatic method to produce such coercions is problematic. It is possible for the user to augment the simplifier via `tellsimp`, essentially giving advice to the internal programs, but this is hardly automatic: it requires rather careful delineation of all needed rules.

But in many cases, the desired result is fairly clear, and these and some more cases the result can simply be "legislated," included in the documentation, and taught to users. Thus in Macsyma, `rat(x)*x` becomes the (rat) result x^2 because the documentation states that the canonical rational expression (CRE) form produced by `rat` is contagious through rational operations like "*".

This is the kind of integration we think of for Level 2.

We can think it through more abstractly, mathematically. If we introduce a new set of forms R, does it correspond to a new abstraction or just a new representation for a known one? Can we store, as declarative information, what operations are required on elements of R? Certainly some of this can be approached systematically (as in Axiom). For Macsyma, this knowledge is not especially declarative; arguably most of it is going to have to be placed in the simplifier, and so is algorithmic, but also somewhat confined. We need to think about which operations are allowed on the new form. If the forms are "new" then corresponding simplification routines can be written. Thus if a function named `csx` analogous in some way to `cos`, we could write a `simpcsx` routine and "solo" simplifications would be restricted to that one program. However, `cos` must also participate in rather complicated ways with `taylor` and `trigsimp`. Perhaps `csx` would need such additional work as well.

If the new domain R allows multiplication and addition, then this require changes to programs like `simptimes`, `simplus`, and perhaps other subroutines to the main simplifier, `simplifya`. These may requiring great care, being patched as alterations to the Lisp code, or can sometimes be handled by `tellsimp`. The interaction of elements of R with the other data types, starting with integers, must be written out. It becomes evident in writing such programs that one must make a collection of often arbitrary decisions in coordinating the new representation with the old, including how to deal with errors, coercions, and various mixtures. These are not new thoughts, but have always permeated Macsyma. One set of decisions that cropped up early was how to deal with combinations with floating-point operands when the other operands are exact. What precision should be used for calculations?

It is necessary to provide some tagging information for the new data types, usually consisting of a header word or two. Most of the Lisp programmed operations look at this header before proceeding further. For example, $a + b$, which in Lisp might look like `(+ a b)` would be, in Macsyma internal form, a variant like `((mplus simp) $a $b)`

The rational form looks like `((MRAT`SIMP (ab) (%I2 %I3)) (%I2 1 1 0 (%I3 1 1)) . 1). Our new "PR" form might look like `((PR ...)...)` where the ... are not easily written out. Nevertheless, the simplifier is informed from the header "PR" that the data is of the special pole-residue type. In the absence of any instructions on how to simplify or combine it with other material, the simplifier will treat an unknown header as a function name, and try to just simplify its arguments and carry it along. In this case it is imperative to keep the simplifier out of the middle of such forms which are not even Lisp lists. Therefore a `simp-pr` program may be needed to handle such issues. A one-line program "how to simplify PR-expressions" should be defined as an "identity" function, returning the expression unchanged. (In fact, the Macsyma simplifier doesn't delve into Lisp "structures" and so the PR structure is never touched. If we had made pieces out of Lisp lists, we would definitely have to take some steps to keep the simplifier in line.)

One way to do this is to define the PR "macsyma function as an MFEXPR*" with definition (LAMBDA (X) X).

What should we do if a user creates a PR form and tries to add an ordinary expression to it? We could take any PR form and add 34 to it. We cannot add a new variable in (PR forms allow only one variable), nor can we use non-rational forms, e.g. adding $\sin(x)$. An error might plausibly result in an "unevaluated"

expression, waiting for (say) the unacceptable parts to be removed by substitution, and then evaluated.

4.3 Aside: Do we really need "headers" for every datum in Macsyma?

What if we used, instead of Lisp, a language with strong typing? Although Lisp allows type declarations, and to some extent implementations generally check them at compile time, the language specification in fact allows an implmentation to ignore them. In a strongly-typed language we don't need explicit type headers. The premise of most such systems is that

1. Significant time and space is consumed by explicit types and type-checking, and so it should be avoided for efficiency.

2. Compile-time type checking can find bugs even without execution—it reduces the need to generate a run-time system and run tests.

It is also plausible if in fact your data types are known in advance and you can compile your whole system with full knowledge of the range of types that will be used. Arguably the dynamic nature of interactive computer algebra systems makes it difficult to know all the types in advance, but setting that aside, consider these arguments:

Checking explicit types is probably expensive if your program is mostly dealing with data types that are very small and are supported immediately by hardware. This is the case for small integers or floats. Software tagging of 32-bit numeric quantities seems space and time inefficient. Tags are less plausibly a problem if your main data types are (for example) matrices, trees, or polynomials, and you plan on allocating (and deallocating) memory for an initially unknown quantity of them.

Thus we can argue that for objects taking more than a few words of memory, using a word for explicit tagging is not going to be a great problem.

The advantages of explicit tags generally include easier examination of run-time debugging information, where the types are always available.

Given that we are biased in favor of explicit tagging on almost everything, what about *other kinds of tags or other kinds of types?* Common Lisp provides more than just lists, and so we can be more expansive than just tagging lists with first elements like "mplus" or "mtimes."

Common Lisp allows for newly defined types (defined via `defstruct`) or a more general approach via classes (defined via `CLOS`). In terms of Macsyma, there is a peculiar advantage. A newly defined type is not a list, but an atom. In most cases Macsyma defers operations on atoms to the underlying Lisp system. Since it is not possible to further decompose or simplify atoms Macsyma doesn't do anything with them. If a structure (containing, say, a pair of numbers representing a real interval) looks like an atom, and the underlying lisp knows how to do arithmetic on the structure representing intervals, Macsyma perhaps doesn't need to know much about intervals as such.

4.4 Level 3

This is the hardest level, requiring full integration.

Right now, if you see a Taylor series in x, you can use the `subst` command and give a value for x. That is because someone went to the trouble of extending the `subst` program for Taylor series. Perhaps "disrepping" the Taylor series to an ordinary expression. (There is a program `ratdisrep` in Macsyma for this purpose.)

However, if you see a Poisson series in x, and you try the subst command to give a value for x, the expression will not change. That is because no one went to the trouble of extending the `subst` program for Poisson series. Substitution can be done on the `outofpois` form or a special `poissubst` program can be used.

Taylor series are more fully integrated into Macsyma, and can be fed into all (or most) programs. Something reasonable should result in each case: The "quick fix" available in many programs is to just check

and "disrep" the Taylor representation. However, this may lose significant advantages, and you may actually want to do something clever. For example, in the Taylor series case, you lose truncation information. You would also lose access to faster differentiation, integration, evaluation, or other manipulation, such as series reversion.

What programs and facilities need to be re-examined in the context of new representations for full integration? A short list to check would have to include all arithmetic routines, diff, subst, solve, display, file input/output. Of course some new representations would have a different short list, e.g. images, vector graphics, sounds. Our previous suggestion of intervals would require (for example) figuring out what to do, if anything, with gcd, factoring, plotting, and other operations – perhaps going through the index of all operations to see what matters.

5 Not everything should be done by mucking with the simplifier

You might really want to have special programs, for example to compute derivatives of PR forms without converting out of PR form, computing the derivative, and the converting back.

It may be more than a convenience; it may be a necessity. For the special bigfloat forms, it is quite important that (say) cosine is computed on the bigfloat, and not some other form that would lose precision. The program to compute the derivative of a Poisson series has slightly different semantics, and is many times faster than the usual differentiation.

In reality, incrementally adding new facilities is potentially time-consuming and error prone, and requires some expertise. Fortunately there are a few examples of how to do it, and new additions might follow the same path.

This rest of this paper gives a example.

6 Incorporating Pole-Residue Forms

Recently we wrote an experimental system for representing rational functions in one variable by means of a collection of poles and residues [1]. The initial implementation of PR-rat forms ignored the issue of serting them into Macsyma, certainly not considering this as a primary design requirement. Essentially we assuming that we could, later, use explicit tagging. We just defined a data structure named `pr-rat` with two components, a "definite part" which is a polynomial: a vector of numeric coefficients, and a "pole/residue" part which is a hash table indexed by poles. For each pole a list of residues of different order is stored. We assume all these PR expressions are in the same particular variable, and so we should record what that might be. We could either use the fact that `pr-rats` are atoms and change the Common Lisp arithmetic on atoms (this kind of overloading of Common Lisp arithmetic is discussed in another paper [2], or we could do what has been done in the past, namely make up a lisp-list form. In this case we made up a header: `((PR variable simp) PRform)`. Next we thought about how to display the PRform. We have a choice. We could alter the Macsyma display program to display it automatically, and go about it several ways.

- Compute a linear format for the whole expression: in particular figure out the width so that it conforms to the expectations of the system display format program `nformat`, and let the display program write out a string. or
- We could augment two programs. The `nformat` program needs a PR-format routine to determine how much space is needed, vertically and horizontally, including markers like parentheses for proper precedence. It takes into account the width available, and so can break expressions up over several lines. The second pass must actually spew out the glyphs in the right place, and so may need to know about PR-format specifics as well. or

- Provide to the underlying Lisp system a display method: this is done by attaching a print method to the object PR-form. The Macsyma formatting system will just echo it as the print-string associated with an atom. This is quite similar in effect to the first option. or
- We could tell Macsyma to keep its mitts off a PR form, and insist that any of them be displayed as an abbreviation, say "PRform". Then to display it, one would have to deliberately select that component of an expression and apply to it a special program we have written, PRDisplay.

The advantage of the first three approaches is that we can display mixtures of material like cos(PRform), and the recursive nature of the display program will "make it all work". The advantage of the last approach is that we can deal with issues under our own control, such as doing clever buffering or expression line-breakup. Knowing that we can display a PR form with the full width of the display can be very helpful in the case of very large expressions. (This is the situation for some Poisson series: Apparently expressions which are very large may still be interesting; we wrote a Macsyma display program to incrementally display these expressions: formatting "the whole thing" before any display would be very memory intensive.)

7 Pole-Residue integration

We are aiming mostly at level 1, here.

In more slightly more detail, assume we have facilities we can convert from a polynomial to a PR representation, and from a (single) pole of order k to a PR representation. We can take a list of programs like this: `pr-plus, pr-times, pr-print, poly-list-of-coefs-to-PR, pole-and-residue-to-PR.` and build Macsyma-level interfaces by simply prefacing a Lisp function's name with a $.

```
(defun $PolyToPR(elist var) (list (list 'pr var 'simp)
  (list (list 'pr var 'simp)(poly-list-of-coefs-to-PR e))))

(defun $FromPR(pr) (pr-to-ratio pr (cadar pr)))

(defun $PRTimes(e1 e2)
  (list (list 'pr var 'simp) (pr-times (cadr e1)(cadr e2))))

(defun $PRPlus(e1 e2)
  (list (list 'pr var 'simp) (pr-plus (cadr e1)(cadr e2))))

(defun $PrintPR(p) (pr-print (cadr p) (cadar p)))
  ;; provide the var name to print
```

We have elided various details from these programs, especially error-checking, but the full source is available.

Some other notes on the program: we extracted and re-wrote parts of the polynomial zero-finding program, ALLROOTS, naming it MYROOTS. When we loaded the PR programs into Macsyma, we expected to not have to use our own zero-finding code (which was, after all, derived from the Macsyma source!). Unfortunately, we had forgotten a complication. The concept of complex number in Lisp post-dates the writing of Macsyma, and so ALLROOTS answers look like expressions containing multiplication (by a symbol representing i) and an addition, e.g. $x = 3.2 * i + 5.0$ rather than atomic numeric complex values, e.g. `#c(5.0 3.2)` in Common Lisp. Since the rest of PR knows only about Common Lisp numbers and not `((MPLUS SIMP) 5.0 ((MTIMES SIMP) 3.2 $%I))` we decided to just use our MYROOTS program.

This in turn leads to another problem. Macsyma needs to find an order among all objects in its class of expressions, in particular so that it can find a preferred form: is it $a + b$ or $b + a$? Given two numbers, it naturally uses Lisp's built-in predicate, >. Unfortunately this doesn't work for complex numbers which do not form an ordered field. Since we were guilty of adding Common Lisp complex numbers to the Macsyma expressions, we must also patch the program that sorts such things. It is called "GREAT" and since the Maxima version is open-source we can fix it. The fix is in the source comments file for PR. (polemac.cl). But inserting this fix merely moves the problems to another program where the failure to accomodate Lisp complex numbers appears. The simplification of a number times "complex 0" fails. We have two alternatives:

1. Continue to push complex numbers into Macsyma code, making sure that all tests for "numberp" do not also assume "realp". That means any numeric comparisons by ">" and its relatives must be guarded in some way. Also, algorithms like integer GCD must be revisited, and canonical forms for complex rationals must be specified (rationalizing denominators). All numerical programs from log, exp, trig, to special functions need examination. Or

2. We can convert our result back to the old form used in Macsyma, using the complex unit as though it were a symbol.

This is not the only place there are such issues in Macsyma. Common Lisp directly supports exact rational numbers. Macsyma assumes only exact integers from the underlying Lisp, and so has had, since the original design in 1967, another encoding, as a list, of rational numbers (internally, 1/2 is `((rat) 1 2)`). Which should be used?

For our own purposes we would like to use actual CL complex numbers as well as actual CL rationals, but this requires (essentially) debugging an unknown number of programs, most in the simplifier, but also in the display, and possibly in other areas of the system as indicated above.

It may be instructive to pursue whether `tellsimp` can be used to place `PRPlus` and `PRTimes` in (all) the right places. Consider this:

```
matchdeclare([pa,pb],isPR)$
isPR(p):=not(atom(p)) and part(p,0)=?pr$
tellsimp(a*b,PRTimes(a,b))$
tellsimp(a+b,PRPlus(a,b))$
```

In fact this code will produce useful effects, but it does not determine how to deal with the product or sum of PR forms and other forms, nor how to deal with a product of three or more forms. It does not give guidance as to how to deal with PR forms with different variables. (Recall that the forms are useful only with univariate rational functions). The use of `tellsimp` also will slow down operation of Macsyma, though whether this is perceptible or not depends on how the system is being used.

Returning to the display issue: if it seems sensible to put together a 2-dimensional display of PR forms, one can set the "DIMENSION" property of PR to the name of a formatting program for PR forms. The kind of arguments and computation needed can be explored shown by tracing some of the pre-exisitng DIM-functions.

(for any operator p like MPLUS or %INTEGRATE, look at the properties by (symbol-plist p), and you will see a host of properties having to do with simplification, evaluation, parsing, display, producing a TEX form, compilation, and other matters.)

8 Consequences

Considering the complications of interfacing (see `polemac.cl` for details) of a new package to Macsyma, including what must be characterized as unforeseeable complications (e.g. complex numbers!) what have we accomplished?

- With proper engineering, presumably with some more error checking, documentation, and tuning, the experimental facilities can be used by any Macsyma programmer.
- Direct comparisons can be made between the new package and existing packages, at least when they are both appropriate. For example, the results can be compared as checks. They can be compared on timing and storage utilization. A level playing field with respect to subsidiary activities (in this case, bignum arithmetic), is more easily determined.

9 Examples for PR

```
(c10) prpole(3,4,5,x);
(d10)                              pr(3/(X-4)^5)
(c11) frompr(d10);
                                        3
(d10)             ------------------------------------------
                   5       4        3        2
                  x  - 20 x  + 160 x  - 640 x  + 1280 x - 1024

(c11) factor(d10);
                                      3
(d11)                               --------
                                           5
                                    (x - 4)
```

As another example, the expression $x^2 + 4 + 1/(x^2 + 5)$ can be re-formed (here, using single-precision polynomial rootfinding) as the symbolic expression

```
prplus(prplus(prpoly([4, 0, 1], x),
       prpole( 0.22361*%i, -2.23607*%i, 1, x)),
       prpole(-0.22361*%i,  2.23607*%i, 1, x)).
```

Re-doing this, but using double-precision, and carrying out the evaluation, gives

```
pr(#c(0.0d0 -0.2236067977499789 4d0)/(x-#c(0.0d0 2.23606797749979d0)) +
   #c(0.0d0 0.2236067977499789 4d0)/(x-#c(0.0d0 -2.23606797749979d0)) +
   4 + 0*X + 1*X^2)
```

Conversion back to conventional form yields

```
(x^4+9*x^2+ 21.000000000000004d0)/(X^2+5.000000000000001d0)
```

which is close to the correct

```
(x^4+9*x^2+21)/(x^2+5).
```

Note that if the coefficients are themselves compact, a displayed canonical pr form
`pr((3/(X-4)^5 + 34/(X-5)^8 + 22/(X-5)^6))`
can be rather compact too, compared to most alternatives. On the other hand, for some expressions, the display of each of the polynomial zeroes to 16 decimal digits can be quite bulky. *This does not necessarily reflect an increase in the actual memory which is needed for storage*; storing a polynomial by a vector of explicit exact zeroes should be of the same order of storage as a vector of coefficients of explicit floats. Naturally a sparse coefficient vector may save space, but pr vectors with multiple zeros may save space too.

10 Conclusion

There are substantial barriers to integrating new data structures for mathematical abstractions in Macsyma. The difficulties have been confronted repeatedly, and in some cases, successfully. While some issues can be resolved by careful thought and use of declarative knowledge in a programming framework addressing math and abstractions, not all are so easily resolved in isolation. Some additions require re-thinking previously settled matters, and some require making fairly arbitrary choices. (You might think that the product of two non-zero items is non-zero, until someone sets up arithmetic to be "modulo 6", where 2 times 3 is 0. You might think that zero times anything is zero until someone introduces "infinity".)

The lessons of Macsyma were not particularly observed in the design of later CAS, and so the problems were simply reproduced rather than remedied. We have not seen the "silver bullet" that could fix such problems by any broad strokes. We hope that the examples elaborated will provide grist for checking out the design of CAS. Our evidence is that merely waving buzz-phrases like "object-oriented" at the problems do not make them go away.

References

[1] Richard Fateman. "Computing with poles and residues" draft(tinyurl.com/44jdf/poles.pdf) submitted for publication. March, 2005.

[2] Richard Fateman. "Building Algebra Systems by Overloading Lisp" submitted for publication.