A Review of Macsyma

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

1982-1984, revised lightly 10/2001

Abstract

We review the successes and failures of the Macsyma algebraic manipulation system from the point of view of one of the original contributors. We provide a retrospective examination of some of the controversial ideas that worked, and some that did not. We consider input/output, language semantics, data types, pattern matching, knowledge-adjunction, mathematical semantics, the user community, and software engineering. We also comment on the porting of this system to a variety of computing systems, and possible future directions for algebraic manipulation system-building.

1 Introduction

2 Overview: The achievements, the problems

The Macsyma[1] algebraic manipulation system grew in part from observing the shortcomings of Mathlab [7], and other earlier systems. Macsyma prospered by incorporating the new algebraic algorithms of the late sixties and seventies, and by taking advantage of better and lower-cost computer hardware and software. Looking back nearly two decades towards its first appearance, we feel that it is appropriate to go beyond the stage of summarizing or advertising its capabilities, and judge its success.[2] This paper is intended to be a selective discussion: an appreciation of the achievements as well as the shortcomings of Macsyma. Our purpose is to provide a perspective on Macsyma so that, as new systems are

[1](An acronym for "Project MAC's Symbolic MAnipulator"). Macsyma is a registered trademark. This paper is a lightly edited version of a paper of the same name first published in *IEEE Trans. Knowl. Eng. vol 1* no 1.

[2]This paper is not a substitute for a general survey (see, for example, [30]).

created, they re-create the successes, but not the mistakes, of the Macsyma design.

Comments in this paper are necessarily opinions and observations of the author, who was involved in the "Mathlab Group" at the Massachusetts Institute of Technology from 1968-71 as a researcher/graduate student, and from 1971-74 as faculty, and has been active in using algebraic manipulation systems at the University of California at Berkeley, since 1974. Other participants in the Macsyma effort may not share these opinions, but we hope these notes will be thought-provoking to persons involved with building or using Macsyma or other algebraic systems.

After a brief overview of the achievements and problems, we proceed by itemizing various features: within each sub-section we try to give an appraisal of how those features worked. In general when we refer to the Macsyma system it is the version defined by the code released by MIT to the National Energy Software Center in 1982; the same code was also released to Symbolics, Inc. for commerical development. Since various changes have been made to each of these systems, some minor details may differ. On the other hand, the basic outlines and most of the code in all of the Macsyma descendants appears to be the same; indeed, most of it appears to be largely unaltered since 1974 or earlier.

We believe that the single major achievement of Macsyma was in making such a large number of features available in one system. This was composed of many detailed implementations of algorithms or other features in which Macsyma represented the state of the art at the time. In many instances, newer or alternative systems (e.g. Reduce, SAC-2/ALDES, Mumath, Scratchpad-II, SMP, Maple, etc.) have adopted the same approaches and algorithms. Occasionally the new systems have exceeded Macsyma's capabilities within selected subsets of problems.

It would seem reasonable to expect the successes to be easily observable from the literature on Macsyma. (The major reference for Macsyma, the Macsyma Reference Manual [25] is a compendium of most of the commands in the system. Another useful paper is Joel Moses's early survey [28]. A collection of application areas and a summary of the system is provided by Pavelle and Wang ([30])) Unfortunately, published descriptions of Macsyma tend to be only partial characterizations of the underlying complex programs. One cannot generally tell from the manual or survey articles or solved examples, if Macsyma will find a closed form for a particular difficult integral, or if `solve` will solve some problem, or, for nearly any algorithm, how the run-time is related to the size or complexity.

In fact, these inadequate characterizations can cause confusion when (as happens occasionally), someone presents a program, perhaps using a new programming language or "rule-based expert system" technology, and demonstrates that for some sample problems, the simple program has the same apparent documented capability as Macsyma. Does it exceed in all respects the corresponding

facility in Macsyma? It is difficult to tell.

The areas of failure in Macsyma are harder to write about because they often are associated with problems which appear to have no simple resolutions. For the most part, more recently developed systems have not been impressively successful in solving these same problems either. To that extent, the failures of Macsyma point to frontiers for system builders. For this reason, many of the issues, such as the design methodologies (or lack of them), the "artificial intelligence" goals, the mathematical communication issues, and others, are worthy of some analysis. It has seemed too easy for others – for lack of such analysis – to adopt Macsyma's approach without re-appraisal.

In summary, for this review we are not going to attempt to correct deficiencies in the descriptive literature on Macsyma: we will not give a detailed characterization of Macsyma or even an enumeration of features. We will concentrate on a few significant Macsyma components which have not received much detailed attention in the open literature; we hope to provide a survey of the lessons of their successes and failures.

3 Comments on selected features

In each of the sections below we describe a group of features which appear to us as distinguished either in success, failure, or surprisingly often, both.

3.1 Input

The parser and the user language

Macsyma '82 had a top-level user programming language which was intended to be Algol-60-like, and was supported by an elaborate interpreter, which provided the users' window into the system capabilities. The language implementation was based on a top-down precedence parser due to Vaughan Pratt. This technique is exhibited in a purer form in the CGOL [31] system, which is itself implemented in LISP. I believe the current Reduce parser uses a similar strategy. It is an appealing technology and was used in the "Mellowspeak/Scarab" parser written by Soiffer at UCB, and used by Harlan Seymour's "Conform" conformal mapping system [32].

It supported the appealing concept of making the parser extensible by the user. With some modest effort, it could provide parsing of a variety of new syntactic constructions. In practice, syntactic extension was not attempted by most users. I believe that the conceptual burden on the user of converting his notation to the usual functional notation supported by Macsyma was not a major barrier. Most users who believed that Macsyma would solve a hard problem if they adapted their notation slightly, were generally willing to do so. (One mathematician who believed that without proper notation nothing

could be done, found the use of an ASCII keyboard an insurmountable problem. He was, fortunately, an exception.) When a complete "package" was written, sometimes it would be re-engineered for a user community by providing a special syntax.

Although common prefix, postfix, and binary infix operators could be added, some kinds of extensions could not be handled by the parser-extension technique. Probably the bigesst limitation was that expressions which consisted of *several* key words or symbols interspersed among operands could not be added. Other syntactic extensions, including optional and unevaluated arguments to functions, and macro-style definitions were added over the years.

A rather useful feature of the parser scheme was that extensions which were made available for input, were also made available for output. This is described in the section on the display.

While the parser had fulfilled its major requirement of handling routine programming requests of users, the language and parser design had some major failings.

Some failings were caused by the relatively naive parser implementation. These were not inherent in the design, and may have been corrected by recent developments in Macsyma versions available from NESC and Symbolics Inc. Among these complaints were the mediocre error diagnosis, the difficulty in correcting typing errors, and inconsistencies in usage of input from various sources.

More significant problems were present, though. One was the substantial hazard that any extension of the syntax would run into difficulties if used in conjunction with other unanticipated user-written extensions. In particular, given the bounds of the ASCII character set and the limited number of unused single-character operator-like symbols, most users chose the same symbols for "new" operations (e.g. @, %, ~). Multi-character symbols could be used, however. Macsyma's language being inherently type-less, made the overloading of existing symbols (say + or *) quite difficult. Use of overloading and generic operators sometimes could be accomplished by pattern matching, but the much deeper understanding that we have of such issues today was not available in 1968. The advantages obtained by assigning types (and hence sets of operations) to values, was not made available to the user. In fact, the kind of data-directed programming that has been advocated by (for example) Abelson and Sussman [1], was just barely implicit in some of the system programs. I feel that object-oriented programming languages of today, and even aspects of the Ada language design should be influential in further algebra-system developments. (see, for example, The Scratchpad-II [5] language design.)

The language semantics

The semantics of programming languages are usually nominally treated separately from the syntax. In the case of Macsyma, this was certainly the case. There are many variations possible in a language which has to deal with symbols, values, indeterminates, types, and a substrate in which programs and data are interchangeable. The semantics of Lisp are still under discussion, and Macsyma's semantics include Lisp's as a subset.

In its favor, Macsyma's language worked for simple commands and simple expressions. It did not require non-intuitive (non-mathematically relevant) notations in these cases. For more ambitious tasks, the mathematical style was generally lost in the need to construct an imperative-style program.

For those familiar with Macsyma, we provide the following itemization:

- There are, tangled within the language semantics, various pieces of mathematical semantics, and parts of LISP: in particular, the use of lists as primary data structures meant that `cons, first`, and `rest` and dynamic scoping of variables fell out of the LISP heritage. As the result of lengthy group discussions (some led by Eric Osman, Jeffrey Golden, and Joel Moses, I believe) an attempt was made to separate language names into 'nouns' and 'verbs'. An overly simple explanation by way of an example, may suffice: Consider `diff` in the command `diff(y,x)` the derivative of y with respect to x. Here `diff` is a verb: the command results in the value 0, since y apparently does not depend on x. The noun form of `diff` used in the command `'diff(y,x)` is a way of expressing the form of a derivative $\frac{\partial y}{\partial x}$, as might appear in a differential equation. In the context of the discussions in the Macsyma group, the idea of having different noun and verb forms for essentially all commands carried the day, and led to the introduction of name-conversion declaratives and functions like `nounify` or `verbify`. Most of these distinctions do not appear in the printed form of the expressions.

- The structure of type declarations (optional and haphazard) was difficult to explain, or for the system to make correct use of. These declarations were supposed to be used by an optimizing compiler which would make Macsyma's language suitable even for floating-point computation. Because of the complex operational semantics of Macsyma, this translator and compiler were complicated and unreliable[3].

- Dealing with operators in a uniform way was never incorporated in the language model. The default technique for handling any notation that looked like functional application was to treat it as functional application. Any other formalization of operators led into uncharted syntactic and semantic territory. (Notation for "differentiation by x" is in this category).

[3]They have been refined by recent activity (post-1984), however.

This is an important and non-obvious issue about which several papers have been written (see the papers by Golden and Gonnet [14, 15], for recent work). The massaging of expression trees which Macsyma was expert in, was not, in the case of operator calculations, the mathematically most appropriate model for computation.

Several papers in the 1977 Macsyma Users Conference Proceedings [29, 13] provide more discussion of the options for semantics than is possible here. Macsyma suffered from having so much of a system's semantics controlled by global flags, many of which control different parts of the same operation. Since the setting of flags may have been changed by packages loaded in by a hapless user, libraries had to be developed and used with great caution. Indeed, because the syntax was extensible, the syntactic and semantic correctness of a program could be altered by a previously loaded file[4].

Another problem with semantics was documentation. Perhaps one could argue that in the early days, the inadequate documentation of some commands was a feature of the system: when a program was changed it was unnecessary to update the documentation. Put another way, any undocumented feature could be changed without notice. Thus by not saying how factoring, integration, or simplification was done, one could change the algorithm without warning. By being imprecise about how certain simplifications would turn out, the programs could be made more (or less) powerful than they were in the past, without warning. This sometimes caused consternation on the the part of users, who found that in the newest version of Macsyma, their programs either work much better, or much worse, than before.

The tradition of incomplete system documentation continues in The MIT-LCS Macsyma Reference Manual [25] (version 10) of 1983, and its descendents such as the Symbolics (version 11, of 1985) still have numerous only-partially-described programs including those for limit calculations, Taylor series, integration (definite and indefinite), solve, and simplification. By saying almost nothing about space or time performance of algorithms, arbitrarily (in)expensive drop-in replacements for modules in common use, could, and on a few occasions, were, inserted. Part of my own experience in working at MIT included inserting variations of the modular polynomial GCD algorithm. In the case of the GCD algorithm, a switch was installed to select the method used. What was changed from time to time was the default setting.

Prospects for the future: the Language

In fact, the major problem that should have been addressed by the input system was *How do we add mathematical knowledge to a symbolic mathematics*

[4]There are, of course, people who love this kind of environment, and many of them use LISP.

system? The answer which was provided followed along the lines that an *ad hoc* language based on an Algol-ish notation, with associated pattern-matching facilities (discussed later) and a complex semantics, would do. The answer was, in my view, a shotgun approach which could serve only at a relatively superficial level, for communicating serious mathematics.[5] It did not scale up efficiently. Persons using Macsyma as a base for adding serious new mathematics to a computer system either got bogged down in the user-level language, or broke free (somewhat) by writing in Lisp. The user-level language represented, for many, a barrier.

Is there a better answer? Several possibilities come to mind, including a model of communicating "active agents" encoding information as procedures or data [1]. I also believe that it would help to pay more attention to application needs: to emphasize tools for expression of applied mathematics in semantically clear notations, rather than superficially "user-friendly" programs with heuristic-based semantics. An example of this approach is the FIDIL (Finite Difference Language) work of P. Colella and P. Hilfinger at UC Berkeley. Finally, Macsyma, and similar systems tend to emphasize a view of mathematics as the formal manipulation of prefix Lisp-encoded expression trees. A more modern view is that we reconsider and treat our work as the manipulation of elements of an algebraic hierarchy. For this purpose we define and use generic operations based on the "modern algebra" of the data/mathematics we compute with [12]. We believe that this is generally an overly-simplified view, unable to easily represent important concepts from analysis, geometry, multi-valued functions, etc. and their inter-relationships. Yet it is a step in the right direction.

3.2 Display

Macsyma's display program underwent several revisions since its first prototype in 1967. Its latest version embodied about the best layout we have seen for fixed-width font typesetting at an acceptable cost. The program was sufficiently flexible that it could handle, in principle, multiple font sizes and higher resolution positioning, although in deference to the large number of users on conventional terminals, nothing much was done with variable size characters, except for a few experiments and an 'offline' phototypesetter system by Foderaro [foder78] using the UNIX operating system's "eqn" program. It has only been relatively recently (post-1984) that reasonable-speed 2-D variable-font interactive typeset display on workstations has been incorporated in Macsyma-based systems. (For example, [Foster, 84], describes such a system for a SUN Microsystems computer). Symbolics Inc. now advertises a type-setting capability similar to Foderaro's but using the TeX [knuth84] text-processing system. An independently developed TeX system has also been used here at UC Berkeley.

[5] One could argue that this system was "Turing-equivalent" to any other method, but that is hardly convincing. If it were, we could consider writing in Turing machine code, too.

The real problem in display turns out to be the breaking-up of large expressions over multiple lines. Methods for computing appropriate break-points are partly based on typesetters' rules of thumb that are not clearly formulated. The algorithm behind the Macsyma '82 program avoided the potentially exponential costs in finding an absolutely optimal break-up, at the (slight) risk of non-optimal layout. This problem has been studied in other contexts in reformatting the listings of nested programs written LISP, PL/I and Pascal. See also the discussion and references in section 18 of [23] for mathematical typesetting specifically.

As mentioned in the previous section, the display program also takes significant advantage of the parsing mechanism cited earlier. The same information used by the Pratt parser for parsing the user text input into an internal LISP tree is used to reverse the process and display the expressions in the user-defined syntax.

The principal failure in the display technology is one of omission: not taking advantage of developments since 1968 in hardware and user-interfaces. Even using the highly-interactive mouse-based MIT-LISP Machine version of Macsyma, by 1982 no interactive system was released for general use.

Projects in improved algebra system interfaces have surfaced in the a number of places, but none "officially" associated with Macsyma. One notable development is the Mathscribe project, a front-end to REDUCE, at Tektronix Research Lab (Beaverton, OR). Mathscribe is in some ways similar to work undertaken at Berkeley. For example, a recent MS project at Berkeley provides for a menu-based command structure on top of typeset-display-based local version of Macsyma. Selection of subexpressions from the screen is based on pointing with a mouse. The contents of the menu depends on the items selected by the user. Unfortunately, MIT has interposed a barrier to sharing of such user-interface code among Macsyma users, and hence such projects are difficult to pursue.

3.3 Internal structures

Conjunction of data types

Macsyma was probably unique in the extent to which a multiplicity of data types were used together. Conjunction of functional forms, mathematical expressions, canonical forms for polynomials (factored, expanded, recursive), rational functions, Poisson series, Taylor series, floating-point and big-float numbers, exact integer and exact rational numbers, were all possible. This made it possible to express some rather esoteric algorithms as merely translations from one form to another. Macsyma's major precursor in this regard was Mathlab 68, in which a canonical rational form was used for some internal routines, but not made explicitly available to the user. One of Mathlab's most powerful routines was the rational simplication function, which converted an expression into and then out

of a canonical internal form. This was preserved in Macsyma as the RATSIMP command.

Other systems past and present, generally used only one form for canonical simplification; celestial mechanics systems often use some variation of Poisson series; various systems used rational or polynomial canonical forms as their sole representation.

Surprisingly useful were some of the simpler functionalities in this hierarchy. Exact rational and integer arithmetic facilities were widely used for applications, but more importantly, they had a pervasive influence on the design of Macsyma's algorithms because they were implemented rather efficiently in the underlying LISP dialects. The fact that Macsyma (and for that matter most of its contemporaries) supported rational arithmetic (without the introduction of truncation or round-off error) meant that it was sometimes unnecessary to consider error analysis at all. In some cases, the error analysis was removed from the center-stage of computations, and left to some subsequent stage: For example, one might evaluate an indefinite integral exactly, and only then use approximate floating-point arithmetic in evaluating the resulting formula. For good or ill, the cost for this exactitude was rarely made clear to the user, some of whom had no means of assigning a cost to a computation performed on a remote timeshared computer (the MIT Macsyma Arpanet host). Many relatively naive users did not realize that a huge (and perhaps unnecessary) superstructure was also present during their computations; for example, some users made use of exact arithmetic for computing points in crude plots. The arbitrary precision floating-point numbers were also heavily used in a few applications, we suspect in part as a computation-intensive substitute for some numerical analysis.

The big-floats were relatively more convenient to use in Macsyma than other supporting environments: There were several such as the Fortran subroutine libraries (e.g. MP by Richard Brent [brent78], Super Precision by William T. Wyatt, Jr. [Wyatt76]). They were generally batch-oriented, and were either called from Fortran or used from a pre-processed language that is translated to Fortran.

How has this multiple-representation paradigm affected current thinking? It has not been followed by two more recent systems, SMP and Maple. SMP chose a single representation because the designers thought (wrongly, I believe) that one representation was best; I believe Maple chose a single representation initially for considerations of overall size of the system, uniformity of notation, ease of explanation, and a concern for exploiting relatively straightforward modern software engineering techniques for portability and modularity. Recently reported work in Maple on computation of Grobner bases suggests that researchers are willing to consider alternative data representations if they provide important benefits [czapor86].

On the other hand, a logical outgrowth of this multiple-representation idea is present in the recent work at Berkeley, and the Scratchpad group at IBM Yorktown Heights.

If one looks at the canonical forms in Macsyma as implementations of abstract data types with operations and properties, then the abstraction and mathematical categorization present in Newspeak [foder83], Capsules (R. Zippel) and Andante (D. R. Barton) can be seen as a systematic approach to bringing more order into a larger realm. The properties of the Macsyma canonical forms were never made very explicit, partly because some of the properties were removed by the display program; partly because some of the operations were never described to the user or even the system programmer. Thus the Macsyma code, in some instances, is a precursor to these abstraction techniques.

A reasonably accessible description of an approach very similar to that in Macsyma for this multiple-data type/ algebraic hierarchy can be found in the discussion of generic arithmetic and coercion in chapter 2 of [1]. The discussion in this text simplifies and explains what was done, but falls short of describing the kind of approach taken by the more advanced systems mentioned above. The imposition of compile-time types and polymorphic algorithms as in Scratchpad or Newspeak is a step further.

On the down-side of data structuring, at various times during the development of Macsyma, critics claimed that alternative data representations (generally coded in non-LISP languages) would have led to substantial efficiencies. Most speedups were presumably linear in time and space over encodings in LISP, since they were generally linked-list based anyway. Yet factors of 10 or much more have been demonstrated as achievable, in restricted carefully chosen cases, by recoding in 'C' or assembler.

One aspect of representation that was a major contributor to efficiency is a tag on data to avoid unnecessary resimplification. The LISP-prefix data used as the general internal representation of Macsyma was marked with a `SIMP` flag as being "already simplified". Unfortunately, this technique, (somewhat simplified for presentation here) represented (say) $x + y$ as `((MPLUS SIMP) X Y)` was not as easy to evolve as was hoped. The SIMP tags were intended to prevent re-simplification of sub-expressions (here, X and Y) except in special circumstances. Although it would appear that this tagging would avoid an exponential amount of computation, the notion of "already-simplified" was illusive with respect to changing circumstances. A single bit signifying "simplified" could not carry sufficient information to describe the context under which the expression was simplified. Additional tags representing "factored" or other assertions, were even more difficult to preserve. An alternative approach with generally superior results is to make simplified expressions "unique" by hash-coding, and thus keep the complexity of branching in expressions, down. Among others, the Maple system has used this to good advantage. Experiments at Berkeley by graduate student Carl Ponder have demonstrated that hash coding can be added to some of Macsyma's internal routines to convert them to "memo" functions (Maple's option "remember"). These functions remember their argument-value mapping rather than recomputing the function each time. Two particularly attractive functions for this are differentiation and simplification. Early experiments in

Macsyma along these lines were abandoned perhaps prematurely. (One experiment remembered factors, in an attempt to speed up polynomial factorizations).

Others discouraging facets of the Macsyma data structuring techniques are quite historical in origin: Macsyma was written when vectors and strings were not part of the underlying LISP system. The benefits of functional arguments (now often termed closures), and object-oriented programming were not generally appreciated in 1968. Modular programming techniques, now much in evidence in the Common LISP design, were not widely practiced; information hiding and "packages" could have been used to assist in assembling the efforts of numerous programmers over many years into a single system.

Summary: Data structuring

In summary, then, what Macsyma demonstrated was that multiple representations *can* work, but that some overall structure for controlling their documentation, use, and inter-relationships (such as coercion to a displayable form) would be a good idea. Some of this is better understood now, and we believe that much of it may be accomplished with object-oriented or type-centered programming systems with a mathematical framework. We expect that of the systems now under development, Scratchpad II will provide the best evidence for (and against) the new kinds of type systems.

Hash-coding and "memo" functions are nearly essential, it appears, for exploiting certain efficiencies. Breaking out of the "everything is a linked list" model of concrete representation is also of interest: with better abstraction mechanisms, it should not matter.

3.4 Pattern Matching

There were at least three different pattern matchers in Macsyma '82. The first and oldest one was *Schatchen* (Yiddish for match-maker) written by Joel Moses, for use with his symbolic indefinite integration program, `SIN`, and then modified, along with `SIN`, by R. Grabel, to use Macsyma's internal "tagged" expression-tree forms. This system was later used for other problems, for example by Richard Zippel in programs for the recognition of certain classes of closed-form summations. This pattern matcher was quite flexible, but appeared to be useful only for relatively small patterns designed for matching (in some cases) fairly devious instances of generally small expressions. It was potentially very expensive, since it could require backtracking in the course of matching. Commutativity and collection of terms based on mostly syntactic criteria provided important simplifications of integrand-patterns, and the cost to collect such terms and permute objects was not significant in practice, even though inherently exponential-cost algorithms were used. I believe that because one could rely on seeing small expressions, and the number of patterns used was small, it was practical to use Schatchen uniformly and effectively. The problem of

scaling up to large expressions was not significant because large integrands were unlikely to occur (but when they did, results might be rather slow in arriving). The problem of scaling up to large tables of integrals with hundreds or thousands of patterns did not occur because just a handful of rather general patterns were used. The scope of integrals for which Macsyma was appropriate was not expanded by incorporation of tables, but by extension of the Risch algorithm or other algorithmic methods. Althought significant inroads were made into the the integration of some special functions, the vast majority of such expressions, and forms requiring reduction with parameters, were never included.

It is interesting to note that the earlier symbolic integration program by Slagle (*SAINT*), also relied heavily on pattern matching, a program named ELINST (for elementary instance).

Unfortunately, programmers using these pattern matchers must have a fairly sophisticated understanding of the data representations for the algebra systems, and specifically required that the patterns be written as LISP symbolic expressions.

The second pattern matcher was written by Richard Fateman in an attempt to provide, for the user, a language and support system for augmenting the simplification and general manipulation facilities of Macsyma. This matcher relied less on syntax and more on the semantic nature of rational expressions and components which might involve more complicated "kernels". This program demonstrated that it was possible to add certain kinds of user-written rules into an existing framework in a relatively efficient fashion, by using the same kind of indexing on the main operators as used by Macsyma's simplifier. It appears that at least one version of SMP's matcher (see [greif85]) used a similar approach. Since the matching did not so much depend on the syntactic vagaries of the input form, there was a chance of the pattern working even without detailed access to the data representation. By comparison with matching capabilities in other systems, the facility generally used fewer but more powerful patterns to distinguish members of a complicated class from non-members. As the designer and implementor of this system, I found it disappointing that making good use of this system required substantial effort. This was for several reasons, probably the most significant being the difficulty in understanding the limitations of the semantic matching. A secondary problem was probably related to the complex scoping rules for variables in patterns and predicates.

A compatible but apparently unreleased alternative pattern recognition program written by Michael Genesereth attempted to change the compiled/interpreted balance proposed by Fateman's program (which was intended to produce compiled patterns).

The third matching program was inspired by the Reduce algebraic manipulation system's LET matching. The external description was largely copied because it seemed to be a different and in some ways better balance between utility (efficiency, applicability), and user comprehensibility than either of the other two. This was written by Keith Nishihara as a class project, and is

described in the Macsyma manual. Some users found this more appealing, especially if they had working Reduce programs to use as models.

Judging from the difficulties experienced by most Macsyma users in using pattern matching, the existing facilities were not successful in alleviating certain preconceived and built-in assumptions of the simplification process, nor in altering the system behavior completely. Efficiency was also a problem, since non-terminating or extremely repetitive and ineffective pattern/replacement programs could be constructed. Yet there were a few nice demonstrations of pattern usage and perhaps this is all we can expect.

Certainly a number of people have pounded on this aspect of algebraic manipulation, including Martin Griss (in Reduce), advocating hash-coding, Richard Jenks (in Scratchpad [21]), advocating compilation and tree-structured combination of related rules, and the SMP group, which has provided an interesting language design combining aspects of indexed sets, arrays, functions, and resolution of missing elements by pattern matching.

3.4.1 Summary of matching

Matching is perennially a topic of discussion. (See [3].) For example, the appeal of Prolog as a language for Artificial Intelligence (and in some peoples' view, algebraic manipulation), is probably based, so far as it is possible to make the choice of a programming language rationally, on the power of (mostly syntactic) pattern match and replacement schemes. Since Prolog seems to be missing a number of useful features (any of which can be simulated, though at some cost), it is not clear if Prolog will provide benefit. Some people are hopeful that parallel execution of Prolog will more than recover the efficiency. This remains to be seen.

The importance of pattern matching is well recognized: it seems to be one of the most popular ways of adding non-procedural information to an algebra system. In fact, to many people, the ease with which one can incorporate varied information efficiently into an existing system is the premier test of so-called "expert system" "shells". Treating Macsyma as such a framework, it would seem that it has a variety of partially useful techniques, but it is apparent that the final resolution of pattern-matching has not been found. In particular, papers continue to be written about matching as done, for example, in the SMP system [16], alternatives in Macsyma [3], and in the massive literature on unification, theorem proving, the programming language Prolog, and the consequences of matching, backtracking, etc. (see, for example, [26]). The solution is not in sight, but perhaps pattern matching is just a short-sighted approach. For others, see the next section.

3.5 Paradigms of Knowledge-adjunction

The use of grand phrases in so-called "Expert System" technology inspired the title of this section. The issue, however, was real for Macsyma, and still remains important: how could users insert new information into Macsyma? Other than by associating values and algorithms to various names, or hooks in the system, perhaps by pattern-matching, a facility was implemented to store information in a less program-oriented data base.

Macsyma's "assume" facility allowed users to insert the answers to anticipated questions which would, from time to time, be asked by some computational routine. For example, declaring `assume(x>0)`, in anticipation of some algorithm's asking "is x positive, negative, or zero" would allow Macsyma to continue without user intervention past the point of the query. A modest deductive facility would allow Macsyma program to infer from that same assumption that x=0 is false. This was a first step in a direction rife with hazards. Clearly the mechanism required to compute needed inequalities derivable from the possible statements was not algorithmic, (not decidable, that is), but another complication appeared: the assumptions held only in certain contexts which had to be layered upon each other and interspersed in the binding of names and values throughout Macsyma. Michael Genesereth provided the initial approach to this difficult task in Macsyma. To some extent it worked silently and protected the user from being asked (redundantly) some easy questions. The user often was nevertheless occasionally asked inappropriate (easy, hard, or irrelevant) questions. This may have been caused by not hooking up all programs to the facility, or by the inability of the facility to decide the answers. Probably no other algebraic manipulation system approached this level of sophistication. A careful study of the extent to which this assume facility accomplished its desired objectives has not been done.

On the negative side, the facility provided in Macsyma was, in a word, annoying. It was unreliable both in the sense of having bugs, and also in the sense of being hard to understand. It was good at inferring that if x were an integer, $x+1$ would be an integer. It did not know that if $x > 1$ and x were an integer, then $1/x$ would not be an integer. It was good on deductions entirely based on a type hierarchy or entirely based on simple linear inequalities. To the user, it was not clear what would be inferred. Furthermore, the cost for inferences was not easily bounded. Because it appeared to be more powerful at determining the truth of inequalities, the facility for dealing with comparisons in general was adopted by some programmers as the facility of choice where much simpler, perhaps bug-free facilities could be used. Imagine the use of the elaborate "assumption" database in computing comparisons in the course of processing a 'for' loop: each time the increment is added to the index variable, the end-test is of essentially unbounded expense.

By merging the semantics for mathematical indeterminates and program variables, deductions were sometimes quite puzzling. For example, n is not an

integer, but a symbol. In $\sum_{n=1}^{10} n$, n assumes only integer values.

It is clear that accumulating knowledge in some form is vital to continued growth in capabilities of symbolic computation systems. Activities at UC Berkeley to incorporate more external data-base information may provide some indications of possible approaches. One which we are experimenting with, involves the use of a large read-only file of definite integrals and a special-purpose lookup mechanism based on hash-coding. Undoubtedly other possibilities are worth pursuing.

3.6 Algorithm collection and evaluation

A large number of algorithms for basic algebraic operations were adapted, tested, and in some cases originally constructed, for use by Macsyma. These included programs for numerous polynomial greatest-common-divisor (GCD) routines, polynomial factorization, computation of limits, summations, canonical forms, Taylor series, definite integration, and several versions of indefinite integration. In many cases, algorithms which originated elsewhere (e.g. for fast sparse determinant calculation, resultants, summation, manipulation of factored-form polynomials) were incorporated in Macsyma shortly after (or sometimes before) being described in the literature. This reflected the fact that most of the underlying structure for these algorithms was already present, and the programming personnel were rarely inhibited from inserting new algorithms into the system.

In the evaluation of algorithms, there was often a natural bias toward the most recently written program: it was, after all, the brainchild of an active programmer; it was this one which was still being developed and would be refined until it was faster than the others. It was, however, often the one exhibiting the most bugs: firstly, it was the newest and most experimental, and secondly, the other algorithms, benefitting from non-use, did not reveal their bugs at all! Some variations of algorithms may therefore continue to have harmless bugs indefinitely.

4 User community

In the early days (1968-71) the community consisted almost entirely of Project MAC and the MIT Artificial Intelligence Group faculty, staff and visitors. It grew rapidly when the ARPANET communication network blossomed in the mid-seventies.

The unique-to-MIT DEC-PDP-10 operating system (ITS) although very advanced in some respects, was difficult to fathom outside the local environment. In spite of this and a number of others barriers that remained in the way of "serious users" outside the MIT geographical area, several users produced prodigious amounts of code, both in LISP and in the Algol-60-style top-level language.

Some of these resulted in contributions to the "SHARE" library. These were of uneven quality, and the documentation was often inadequate. While some contributions combined questionable mathematics, algorithms, and programming, others contained useful and original ideas, cleverly implemented and carefully described.

5 Software Engineering

In 1969, in order to put together all pieces of a Macsyma system, it was necessary to get a group of people together to type in commands to a newly created large LISP system, to laboriously assemble in place the numerous files which constituted the system. The files constituting Macsyma did not fit on the small random-access disk, and therefore constructing the system required mounting and dismounting magnetic tapes.

While larger disks played an important role in simplifying system construction, subsequent innovations included a specially constructed compiler script file which could be fed into a LISP system to direct the compilation and loading of a fresh system. Systems for the automated compiling and loading of large systems from LISP scripts may have been inspired by the efforts (primarily by Jeffrey Golden) to accomplish with a few keystrokes, a sequence of system-construction tasks. This facility predated software engineering utilities in the Lisp machines (defsys) as well as the "make" facility in the UNIX operating system. Macsymas constructed at UC on UNIX-based systems make extraordinarily heavy use of the 'make' utility.

When the early Macsyma system grew too large for the virtual address space of the PDP-10, the LISP system was augmented by an automatic loading mechanism to call in the definitions of commands from files. The techniques allowed the sum total of programs written for Macsyma to exceed the 1.2 megabyte PDP-10 system limits. Given the limits of the PDP-6/10 machine architecture at that time, very effective use was made of the address space and between-user sharing. This was not painless. In fact, substantial intellectual effort was needed to partition Macsyma in an appropriate fashion. Although most machines running Macsyma subsequent to the PDP-10 have had very large virtual paged address spaces, the auto-loading facility continues to be used on some systems where the use of address space, even if filled with unused code, has a higher cost than file-system space.

These features contributed to the software engineering movement by providing ideas for software development environments supportive of large Lisp systems, and perhaps other projects.

On the other hand, another goal of software engineering, that of promoting portable software, was not high on the list of objectives for Macsyma.

For a number of years it was hard enough to run Macsyma on its original development system, much less consider porting it to other computers. A

successful porting of Macsyma to a Honeywell Multics mainframe was done, however, by bringing up a nearly identical MacLisp system on that computer, and consequently providing a hospitable environment for Macsyma. Even so, a number of features were not simulated. In many ways the original PDP-6/10 system contained unique concessions to the needs of the Macsyma program and these could not be easily ported elsewhere. This was recognized at MIT, and for many programmers the unique features were thought of as the common environment. This lead to very non-portable constructions, including absolute path-names to files, tendrils into programmers' own personal directories, subprocess activation requiring specific ITS editors, compilers, etc.

Starting in about 1980, the Berkeley "vaxima" system, based on a UNIX operating system environment and a LISP system written primarily in the C programming language, demonstrated that more common, lower-cost general purpose systems could run Macsyma. Developing vaxima forced the re-learning of some of the lessons of the Multics port: it was necessary to decide where to differ from the ITS system. When no entirely common solution could be found, conditional compilation switches were inserted into the source code for Macsyma. With the hope that diversity would allow for creativity on a variety of topics, programmers at Berkeley and MIT worked to identify critical sections for the VAX and what by 1980 included several other systems: LISP Machine LISP, NIL, and the old Multics MacLISP.

Macsyma's source code at Berkeley was kept under a revision control system (RCS). Since the MIT programmers did not subscribe to this system, occasional rather painful merges of MIT changes had to be made. Since the organization of the Macsyma source code was in many files with dependencies on macro-definition files, making coordinated changes was difficult.

Once the code was largely stabilized, the UNIX-based version, at least, could be ported to a variety of machines. In particular, following the porting of a MacLisp-like dialect (Franz Lisp) to the Motorola 68000 machine, Macsyma was demonstrated on machines of several different manufacturers at UC. Now Macsyma also runs on IBM-370 compatible computers (e.g. IBM 370) and several other architectures.

It is interesting to ask, is Macsyma really portable? If so, how did it get that way?

No one looking at the code prior to 1979 would consider it written for portability; yet in retrospect, it seemed to move more easily, on its base of compatible Lisps, than one would expect for over 100,000 lines of code. Although more attention might have been paid in the early days to making it portable (as was done with Reduce), this might have inhibited experimentation on the one primary machine. The trade-off was that the portable Reduce system grew a larger community of system-programming and algorithm contributors; the Macsyma programmers had only one machine, but it was a good one. We believe that those who were fortunate enough to get quality access to the ITS system were able to get more done in less time than more isolated persons with their own

private copies of Reduce.

Now, with the widespread availability of large-address-space machines capable of running Macsyma, most of the merits of the ITS system are available on small systems; with high-speed networking, the difference between two researchers working on one system, and two researchers working on separate systems has largely vanished.

A further step toward portability of Macsyma (or any other large LISP system) seems to be a tasteful conversion to the Common LISP dialect. Not everything in Macsyma can be supported in the currently defined portable core of Common LISP, nor do most Common LISP implementations make as efficient use of machine resources as Macsyma's current host languages. Nevertheless, it is prudent for any programmer working in LISP in Macsyma-related facilities to maintain an eye towards compatibility with Common LISP.

6 Software Engineering

Over the years, the Macsyma software situation was repeatedly, to use the vernacular, 'wedged'. By this we mean the source code and compiler information was left in an inconsistent state, so that a new system "binary" could not be generated, until some subtle bug could be found. The source code was, from the earliest days, written with macro-expansion for open-coding of functions. In the years after about 1974, data abstraction was somewhat non-uniformly inserted into the existing LISP source code, and placed in separate files which the LISP compiler had to read in, to become 'customized' for the Macsyma-compilation environment. Dilemmas occurred when, for example, the macro definitions for the data abstractions were themselves compiled, in an environment which itself required that those abstractions be loaded in a compiled form.

The sensitivity of such an arrangement of bootstrapping to bugs is notorious. Moving to the UNIX operating system at Berkeley, which forced a new community to provide a more systematic control on source code and on programmers, improved the situation somewhat, but did not cure it entirely. Coordination between sites (MIT and Berkeley principally, but also Kent State and to some extent about 50 others) was difficult for technical reasons: except for MIT and Berkeley, most sites were not on the arpa network. Continued modification of the MIT code, sometimes without regard to the operating environment on UNIX systems, caused some difficulties too. It is interesting to observe that deriving a 'complete' listing of just the names of all files relevant to the MIT Macsyma system was not at all trivial in 1978, and was probably not done accurately for several years, if, indeed, it has ever been done!

The PDP-10 was designed with a larger address space than most other machines of its generation, but one that turned out to be filled up in about 5 years of serious programming. This lead to substantial restructuring and tinkering with design of overlays or auto-load files to maintain the possibility of growth

in program space. The lack of user work-space was never really solved on the PDP-10, and many problems could not be run to completion on that machine. The Multics version of Macsyma provided a potentially larger space, yet allocation of sufficient space and time to run large jobs was generally quite expensive (especially considering the subsidized time available on the MIT-MC PDP-10 computer). Obscured by the move between the PDP-10 and Multics was the fact that many computations of a symbolic sort do not scale up comfortably. Improving the computation by one unit might mean doubling the size of memory and multiplying the CPU time by much more. I have a feeling that moving to Multics salvaged very few computations. The MIT LISP Machines also were potential successors to the PDP-10 for running Macsyma, but (at least the early versions) did not perform well for long jobs, for reasons perhaps associated with their lack of adequate garbage collection algorithms. Early tests showed that the performance of Lisp Machines was quite poor unless there was substantial real memory.

The operating system, MIT's ITS time-sharing system, was a mixed blessing. The original flat, and limited-size directory structure of ITS meant that multiple directories had to be used to store the collections of Macsyma source files, object files, backup copies, data, etc; the six-letter file names (with a six-letter or number second name) were also uncomfortable. ITS was in other respects quite advanced, including probably the earliest "transparent" multiple-machine networked file system. It also accomodated to other user requests rather well, and hosted numerous software projects (e.g. the EMACS editor) which benefitted residents of the system.

The lack of security was generally harmless; the installation of new features overall, probably led to a higher level of synergy, but without adequate discussion, warning, or testing, was sometimes counterproductive. Keeping one or two backup copies avoided some of this distress, but did not provide a convenient test-bed for multiply-dependent modules that had to be debugged simultaneously.

Future systems should clearly take advantage of the last 15 years of software engineering and operating system improvements. Separating out the modules and identifying interdependencies could retro-fit some technologies to Macsyma, but to date, most changes have been fairly conservative in this respect.

7 Documentation

User documentation for Macsyma grew to a bulky three-volume set in the version 11 manual written at MIT. Unfortunately much of it was originally written in some haste and not kept up-to-date. A subsequent revision by Symbolics, Inc. remedies a few faults, but still suffers from the "operational semantics" view: the exact meaning of commands is hard to guess without trying them out.

The system documentation is largely lacking, and with the exception of a very few papers, consists of sparse comments in the Lisp source code. In the early days the tools used for editing did not support re-formatting of any comments, reinforcing the natural inclination of programmers to neglect documentation.

8 Inconsistency, Wrongness

If a system allows you to express several inconsistent ideas, it is asking for trouble. Macsyma asks for trouble. It has a very naive idea about how to compute with infinities, multiple-valued functions, and other boundary type conditions such as 0^0 or $\log(-1)$. For example, if one believes that 0^0 is 1, then 0^k (which Macsyma simplifies to 0) is a "delta function" (e.g. `if k=0 then 1 else 0`. If $\sqrt{xsup2}$ is to be anything other than x (say, $\pm x$ or $|x|$), then one must deny the otherwise somewhat useful notion that $(x^a)^b$ is equivalent to x^{ab}. Here are some other commands which can be run through Macsyma, or in slightly changed syntax, through other systems. These demonstrate various pieces of mathematical nonsense. (note that `%` is a shorthand for "the previous expression")

Fallacies and paradoxes, with apologies to Kasner and Newman,[22]

```
(c1) b+a = c;
```

(d1) $$b + a = c$$

Multiply both sides by $a + b$

```
(c2) expand(%*(b+a));
```

(d2) $$b^2 + 2\,a\,b + a^2 = b\,c + a\,c$$

Subtract the same quantity from each side.

```
(c3) -a*c-b^2-a*b+%;
```

(d3) $$-a\,c + a\,b + a^2 = b\,c - b^2 - a\,b$$

Now factor each side.

```
(c4) map(factor,%);
```

(d4) $$-a\ (c - b - a) = b\ (c - b - a)$$

Now divide each side by the same quantity.

```
(c5) %/(c-b-a);
```

(d5) $$-a = b$$

Now add a to each side.

```
(c6) a+%;
```

(d6) $$0 = b + a$$

QED: the sum of any two numbers $a + b$ is zero.

A proof that $x = x + 1$

The following equation is valid for all n.

```
(c7) eq2:(-(2*n+1)/2+n+1)^2 = (n-(2*n+1)/2)^2;
```

(d7) $$\left(-\frac{2\,n+1}{2}+n+1\right)^2 = \left(n-\frac{2\,n+1}{2}\right)^2$$

We can show it is valid by expansion:

```
(c8) expand(eq2);
```

(d8) $$\frac{1}{4} = \frac{1}{4}$$

Now substitute the same value on each side.

```
(c9) subst(-a,(2*n+1)/2,eq2);
```

(d9) $$(n+a+1)^2 = (n+a)^2$$

Take square roots of each side.

```
(c10) sqrt(%);
```

(d10) $$n + a + 1 = n + a$$

Since $n + a$ is arbitrary, let us give it a name, say x.

```
(c11) subst(x-a,n,%);
```

(d11) $$x + 1 = x$$

QED: $x = x + 1$.

Here's a proof that $\log(-1) = 0$.

```
(c12) expand((y-1)^2) = (y-1)^2;
```

(d12) $$y^2 - 2\,y + 1 = (y-1)^2$$

```
(c13) map(log,%);
```

(d13) $$\left(\log y^2 - 2\,y + 1\right) = 2\,\left(\log y - 1\right)$$

```
(c14) subst(x,log(y-1),%);
```

(d14) $$\left(\log y^2 - 2\,y + 1\right) = 2\,x$$

Now let $y = 0$.

```
(c15) subst(0,y,%);
```

(d15) $$0 = 2\,x$$

QED: x, which is the same as $\log(-1)$, is zero.

Here's a final example that shows that for any function $f(z)$ analytic at the origin (i.e. representable as a power series in z), $f(0) = 0$.

```
(c16) subst(0,z,sum(a[i]*z^i,i,0,inf));
```

(d16) $$0$$

Many additional examples can, of course, be constructed from any of these anomalies. Here is one more: Macsyma responds to the command `solve(x^n-1=0,x)` with `[x=1]` with multiplicity 1. It is unfortunate that this answer (which is produced for arbitrary unspecified n) is incorrect unless $n = \pm 1$.

9 Non-modularity, non-extensibility

Compared to recent efforts at (for example) IBM Yorktown Heights, on the Scratchpad II project, or at UC Berkeley on the SCARAB project, most of the data choices made by Macsyma programmers were cast in concrete. Data abstraction as a technique was not as well appreciated or prevalent. Some choices of data structure were made at a macro-expansion phase of LISP interpretation or compilation: for example, the determination of coefficient arithmetic in the polynomial package. However, other choices were interwoven in the program text dealing with polynomials: that the coefficients commute, and have no zero divisors, for example, was made at several unmarked places. Introduction of "weighted" variables, as used in the `ratweight` facilities: a simple form of

truncated power series, required more than just re-definition of the coefficient macros.

Common concepts were not implemented in shared modules; for example several predicates to test if an expression depends on a variable were independently constructed. The programs `free, freevar, freeof,` and `depends` all are quite similar. Surprisingly, because of the interactions of various larger modules, the integration program uses all of these.

10 Inefficiency on large expressions

At various times it had been proposed that a suitable project for a student would be the development of a systematic technique for the storage and manipulation of expressions which exceed by orders of magnitude the physical memory of the host computer. This turns out to hinge in part, on simplification questions: how does one identify expressions which might be combined (e.g. $3x$ and $5x$) by some "closeness" metric? Macsyma tried very hard to make this metric correspond to some intuitive one. It judged $\sin x$ and $\cos x$ closer than $\sin x$ and $\sin y$. Because of this attention to arguments in ordering, some algorithms in simplications were relatively slow. Hash coding, used in SMP and Maple, is much faster, although it provides only an artificial closeness based on non-intuitive criteria for grouping common terms. Although the Macsyma design included techniques for avoiding the repeated simplification of subexpressions by the use of an "already simplified" flag, and attempted to share common subexpressions, the intent of the design was sometimes circumvented by programmers who felt that the changes wrought by their algorithms required resimplification from scratch. Programmers were usually not immediately concerned with efficiencies on large problems, since most test examples were small in size, and the gross sizes of the expressions needed to reveal the waste were probably difficult to deal with on the PDP-10. Some categories of bugs could not be exhibited on the PDP-10, because the appropriate programs could not all be loaded into memory at the same time.

Although large problems could sometimes be solved by deliberate "staging" of pieces on disk files, Macsyma never really exploited an efficient external model of data such as has been used by typical celestial mechanics programs (hand-packing of disk records, etc). The possible external forms: straight text "batch files", LISP data as ASCII text (with some recognition of shared components) and "fasl" or fast-loading object-code files, were all mirrors of the internal LISP linked-list data format. Additional techniques for saving and restoring of strings, vectors, arrays, and other kinds of packed data were not generally not exploited. Well-known data-base techniques (B-trees, etc.) did not play any role.

One can hope that a more general approach, based on a suitable level of abstraction, perhaps on communication of messages between objects whose representation is not easily altered, will emerge for the future.

11 Input and Output

Macsyma's language was inadequate to deal with the various issues of multiple files used for input and for output. For example, the `writefile` command opened an (un-named) port, and printing produced by any of the display programs went to that file. No mechanism was available other than descending to the LISP level, for multiple-file output. This would have been convenient for applications which simultaneously wrote Fortran commands to one file, produced a transcript of the screen on another, and wrote backup expressions on a third, and plotting data on another. Macsyma '82 provided several separate output streams, but not under direct user control. Advanced use of input (e.g. a stream from light-pen or mouse) had not been directly made available in the user language, although some experiments suggested this would be a useful feature. Indeed, menus and mouse selection of options has appeared in recent additions to Macsyma at UC Berkeley (on Sun workstations) and at Symbolics (for Lisp machines).

12 Model of user

Macsyma '82 had no consistent, dependable, generally useful model of the user: the closest we can come to characterizing the intent of the designers was that Macsyma should have a structure to support and connect various "parlor trick" programs like the symbolic integrator, factorization, application of the distributive law (expansion), and so forth. For many interesting applications, this is sufficient. In fact, one could ask, why raise this issue of a model of the user? Is this an issue in other programming languages or systems? Actually, I think it is: modern dialects of Fortran, C, Pascal, and probably Lisp, all have an underlying model of what is to be accomplished. In some cases, BASIC, for example, the model has changed over the years. Operating systems such as VM/370, UNIX, and MS/DOS have expected user audiences. Languages without a focus (PL/I comes to mind), tend to be less satisfactory.

Macsyma tended not to follow through on the mathematical ideas that should permeate it. A mathematically attentive reader of the manual would probably be stymied by explanations like that of the logarithm "function" which, one is told, is "the natural logarithm function". The 5 flags which control simplification of the logarithm provide some guidance as to how an expression containing the function name "log" can be manipulated, but leaves unanswered a number of very fundamental questions. For example, is the log multi-valued? What is log(0)? If x is a complex number, or is a variable which ranges over the complex plane, how is its logarithm defined? If x is a floating-point number, will its log be computed? How accurate will a floating-point answer be?

Notions pertaining to domains of computation, (e.g. real, complex), had been included in some isolated areas, but not integrated into the system. The

approach, as it developed over the years, was "If you don't like the behavior, set a flag." This works, for a while, until the interactions of the flags become too burdensome computationally or too daunting for the user to grasp.

13 Interactivity

Macsyma was designed to be interactive, although the initial human interface would undoubtedly shock people today. The display terminals available at MIT in 1968 were extremely primitive by current standards. Eventually the system had to accomodate various printing terminals and use features like tabulation, cursor addressing etc. Especially after the Arpanet community, and its host of TI Silent-700 terminals, came to have a major impact, most efforts to use graphics were constrained to be purely local. At MIT, impressive curve drawing capabilities were possible on the display system used for Prof. A. Bers' plasma physics group.

The idea of pointing at expressions using a light pen was demonstrated in the late 60s by W.A. Martin but was not picked up again until the 1980s when "mice" became popular as workstation input devices. Still, the "official" Macsyma 82 system failed to use this. Subseqent development has included use of such features, but differing hardware and software bases, combined with a divergent of code, has led to a divergence in approaches.

14 Conclusions

Some features in Macsyma worked out fairly well and might be emulated in future systems, or maintained in future versions of Macsyma. Other features should be changed. Absolute objectivity in system design is rarely possible, and in a system as multi-faceted as Macsyma, there are too many influences to easily find a consensus in some matters.

We hope that this sketch of the history of Macsyma, and some admittedly subjective experience and opinions as given in this paper may be useful in deciding future directions in system building.

15 Epilog

In the years after 1979 when the development of VAX Macsyma at Berkeley first made it feasible and inexpensive to run Macsyma off the Arpanet, the community lost some of the cohesion that resulted from the central site. The UNIX 'use-net' and mail through the Arpanet and CSnet was used for the reporting of some bugs electronically. Other bug reporting was done via US mail to the University of California, Berkeley. Reports were relayed, if appropriate, to MIT.

While some users were able to get only occasional updated code on the VAX, others were able to update copies every few months, usually through Berkeley.

In 1982, when MIT sold its rights to Macsyma to Arthur D. Little, Inc., which then contracted with Symbolics, Inc. for distribution and support, aspects of the community changed substantially. The newly proprietary nature of the code provided an impediment to casual interchanges amongst users. The MIT Macsyma Consortium PDP-10 went "off the air" in October, 1983 for users outside MIT.

Between 1983 and the present, in spite of the distribution arrangement between MIT and Symbolics, the Department of Energy pressured MIT to place a copy of the source code for Macsyma in a government distribution center: the National Energy Software Center (NESC) Library at Argonne National Laboratory. The initial version of this code, an implementation of Macsyma in the NIL Lisp dialect, was named DOE-Macsyma. It requires a large VAX/VMS computer system to run, and was restricted, by agreement with MIT, to be available for purchase by end-users only. Modifications of this code were later placed in NESC to run on various Lisp machines running ZetaLisp or TI Explorer Common LISP. Another version of Macsyma, referred to as VAXIMA, and based on joint work of MIT and UC Berkeley, was deposited by UC in the NESC in July, 1986. This copy, an implementation of Macsyma in Franz Lisp, includes source code and executable images for VAX UNIX or ULTRIX systems. Companies including Paradigm Associates of Cambridge, Mass., and Franz Inc., of Berkeley, Calif. have been improving the usability of the NESC's Macsyma code, and providing support for these versions.

By virtue of the availability of Franz LISP on a wide range of processors, VAXIMA has been demonstrated on numerous Motorola 680x0-based systems: SUN Microsystems workstations, Tektronix 4400 systems, Masscomp, Pixel and similar computers since 1983. Franz Inc. supports Macsyma on IBM 370 mainframe systems and a variety of other machines. Pyramid Computers has also demonstrated a Macsyma running on Pyramid's own version of Franz Lisp.

Macsyma runs now on machines which cost under $15,000, where it has performance far superior to its original development environment (A PDP-6/PDP-10 KA system). It also runs on multi-million dollar computers at substantial speeds. It would be beneficial if system developers had low-cost access to the code as well as a working version of the system, since it appears that contemporary system builders are repeating some of the same mistakes we saw Macsyma, and sometimes painfully reproducing its successes. Research in algorithms and system interfaces are hampered by the clumsy distribution mechanism imposed on users by MIT (via NESC or Symbolics). It is unclear to what extent the legacy of Macsyma will emerge in new systems or revisions of Macsyma; we hope this paper helps designers and users of systems distinguish the wheat from the chaff.

(more recently: October, 2001). For some period in the 1990s Macsyma Inc, an independent company acquired the rights to Macsyma from Symbolics, and

revised the code in various ways, including providing a graphical front end for Microsoft Windows machines, enhancing numerical code, and adding various modules and fixing bugs. The program on UNIX was running under a version of Kyoto Common Lisp, and under Windows, CLOE, a Common Lisp originally produced by Symbolics. The company appeared to be losing money steadily, in spite of various papers suggesting its technical superiority to the competition, principally Mathematica and Maple. In early 2001, the assets of the company changed hands, and the new owner seems to have closed the company and taken the software off the market.

Prof. W. Schelter, a mathematics professor at the University of Texas (Austin) who over the years had been supporting a free Macsyma system (called Maxima), received clearance from the US Department of Energy to enhance and distribute the source code of Macsyma. He packaged the system in various ways to use Tcl/Tk as a front end and GCL (Gnu Common Lisp, a system he helped develop from Kyoto Common Lisp). Various additions that were executed and in the planning stages were not completed at the time of Prof. Schelter's untimely death in July, 2001. His project and code continue to be available, and various efforts to form a working group to further enhance the code are operating under the auspices of the University of Texas and/or Sourceforge.

16 Acknowledgments

I would like to thank Prof. Joel Moses of MIT, and Jeffrey P. Golden, and Kent Pittman, formerly of MIT and now of Symbolics for commenting on drafts of this paper. Opinions expressed in this paper are my own, and do not necessarily represent the views of the reviewers or government sponsors. Work reported herein was supported in part by the Army Research Office, grant DAAG29-85-K-0070, through the Center for Pure and Applied Mathematics, University of California, Berkeley.

References

[1] Harold Abelson and Gerald Jay Sussman with Jule Sussman.

Structure and Interpretation of Computer Programs, MIT Press, Cambridge, MA, 1985.

[2] R. P. Brent, "A Fortran Multiple-Precision Arithmetic Package," *ACM Trans. on Math. Softw. 4* no. 1, (March, 1978) (57-70).

[3] Gene Cooperman, "Semantic matcher for Macsyma," in B. W. Char (ed) *Proc. 1986 Symp. on Symbolic and Algeb. Comp.* ACM Univ. Waterloo, Ont. Canada, July 1986. (132-134)

[4] S. R. Czapor and K. O. Geddes, in B. W. Char (ed) *Proc. 1986 Symp. on Symbolic and Algeb. Comp.* ACM Univ. Waterloo, Ont. Canada, July 1986. (233-238).

[5] J.H. Davenport, P. Gianni, R. D. Jenks, V.S. Miller, S.C. Morrison, M. Rothstein, C.J. Sundaresan, R.S. Sutor, B.M. Trager, *New Scratchpad*, Math. Sciences Dept., IBM T.J. Watson Res. Ctr, Yorktown Hts. NY. see also R. D. Jenks and B. M. Trager, "A Primer: 11 Keys to New Scratchpad," *Proc. Eurosam 84, Lecture Notes in Computer Science 174*, Springer-Verlag, 1984.

[6] A. Doohovskoy, "Varieties of operator manipulation," *Proc. of the 1977 MACSYMA Users' Conf.* NASA CP-2012, July, 1977, (473-490).

[7] Carl Engelman. "The Legacy of Mathlab '68" in *Proc. 2nd Symp. on Symb. and Algeb. Manipulation*, (ACM) 1971. (29-41).

[8] Richard J. Fateman. "Macsyma's general simplifier: Philosophy and operation," *Proc. of the 1979 MACSYMA Users' Conf.* Washington, DC. June 20-22, 1979. (563-582)

[9] Richard J. Fateman. "On the Construction of Algebraic Manipulation Systems," (submitted for publication).

[10] R. Fenichel, "An On-line System for Algebraic Manipulation," doctoral dissertation, Harvard University, July, 1966, also Report MAC-TR-35, Project MAC, M.I.T., available from the Clearinghouse, document AD-657-282.

[11] , John K. Foderaro, "Typesetting Macsyma Equations," MS Report, EECS Dep't., Univ. Calif., Berkeley, 1978.

[12] John K. Foderaro, "The Design of a Language for Algebraic Computation Systems," Ph.D. dissertation, EECS Dep't., Univ. Calif., Berkeley, 1983.

[13] Jeffrey P. Golden. The Evaluation of Atomic Variables in Macsyma," in C. M. Andersen (ed) *Proc. 1977 Macsyma Users' Conf.* Univ. of Calif, Berkeley, July 1977. (109-122)

[14] Jeffrey P. Golden. "An Operator Algebra for Macsyma," in B. W. Char (ed) *Proc. 1986 Symp. on Symbolic and Algeb. Comp.* ACM Univ. Waterloo, Ont. Canada, July 1986. (244-246)

[15] Gaston H. Gonnet. "An Implementation of Operators for Symbolic Algebra Systems," in B. W. Char (ed) *Proc. 1986 Symp. on Symbolic and Algeb. Comp.* ACM Univ. Waterloo, Ont. Canada, July 1986. (239-243)

[16] J. M. Greif, "The SMP Pattern Matcher," in B. F. Caviness (ed), *Proc. Eurocal '85, vol. 2,* Lecture Notes in Computer Science 204, Springer-Verlag, 1985, 303-314.

[17] A. C. Hearn, *Reduce 2 User's Manual* University of Utah Computational Physics Group Report No. UCP-19, March 1973.

[18] *Reduce 3 User's Manual* The RAND Corp.

[19] A. C. Hearn, "A new REDUCE model for algebraic simplification," *Proc. 1976 ACM Symposium on Symbolic and Algebraic Computation,* August, 1976, (46-50).

[20] R. D. Jenks, "A pattern compiler," *Proc. 1976 ACM Symposium on Symbolic and Algebraic Computation,* August, 1976, (60-65).

[21] R. D. Jenks, "SCRATCHPAD/360: Reflections on a language design," *SIGSAM Bulletin 13,* no. 1, Feb., 1979, (16-26).

[22] Edward Kasner and James Newman, *Mathematics and the Imagination,* Simon and Shuster, New York, 1940.

[23] , Donald E. Knuth, *The TeXbook, Addison-Wesley, Reading, MA, 1984.*

[24] Knut Korsvold, "On-Line Algebraic Simplify Program," Stanford A.I. Project Memo 37, Nov. 1965, 30 p.

[25] Macsyma Reference Manual*, Lab. for Comp. Sci, MIT, Jan, 1983 (2 volumes: version 10), available also from the National Energy Software Center (NESC), Argonne, IL. Similar manuals are available from Symbolics, Inc., for example, version 11 (Symbolics, Inc.) Oct. 1985.*

[26] K. McIsaac, "Pattern Matching Algebraic Identities," SIGSAM Bulletin, 19, no. 2, (May, 1985).

[27] Joel Moses, "Algebraic simplification, a guide for the perplexed," *Comm. A.C.M. 14,* no. 8, Aug., 1971, (527-538).

[28] Joel Moses, "Macsyma: the Fifth Year," Proc. Eurosam 74, SIGSAM Bull. (105-110)

[29] Jole Moses. "The Variety of Variables in Mathematical Expressions," in C. M. Andersen (ed) *Proc. 1977 Macsyma Users' Conf.* Univ. of Calif, Berkeley, July 1977.

[30] Richard Pavelle and Paul S. Wang. "Macsyma from F to G," *J. Symbolic. Comp. 1,* no. 1, (69-100).

[31] Vaughn R. Pratt, "Top Down Operator Precedence,"

ACM Symposium on Principles of Programming Languages, Boston, MA October, 1973. See also, a detailed memo (1977) "CGOL - An Algebraic Notation for MACLISP Users" distributed with the CGOL source code.

[32] Harlan Seymour. "Conform, a conformal mapping system," MS Project U.C. Berkeley, EECS Dept. 1985.

[33] R. G. Tobey, R. J. Bobrow, and S. N. Zilles. "Automatic Simplification in Formac," *Proc. AFIPS 1965 Fall Joint Comput. Conf.,* (1965) (37-52).

[34] W. T. Wyatt, Jr., D. W. Lozier, and D. J. Orser. "A Portable Extended Precision Arithmetic Package," *ACM Trans. on Math. Softw. 2,* no. 3, Sept., 1976, (209-229).