Symbolic Mathematics System Evaluators

Richard J. Fateman
Computer Sciences Division
Electrical Engineering and Computer Sciences Department
University of California, Berkeley
USA

Abstract

"Evaluation" of expressions and programs in a computer algebra system is central to every system, but inevitably fails to provide complete satisfaction. Here we explain the conflicting requirements, describe some solutions from current systems, and propose alternatives that might be preferable sometimes. We give examples primarily from Axiom, Macsyma, Maple, Mathematica, with passing mention of a few other systems.

1 Introduction

A key issue in the design of computer algebra systems (CAS) is the resolution of what is meant by "evaluation" — of expressions and programs in the embedded programming language of the system.

Roughly speaking, evaluation is a mapping from an object (input) and a specified context or environment to another object that is a simpler or more specific object (output). Example: $2+3$ evaluates to 5. More specifically and somewhat pedantically, in a CAS, evaluation involves the conventional programming-language mapping of variables or names (e.g. x) to their bound values (e.g. 3), and also the mapping of operators (e.g. $+$) to their actions. Less conventionally, CAS evaluation generally requires resolution of situations in which a variable "has no value" but stands only for itself, or in which a variable has a value that is "an expression". For example, given a context where x is bound to 3, y has no binding or is used as a "free variable", and z is $a+2$, a typical CAS would evaluate $x+y+z+1$ to $y+a+5$.

In simple cases this model is intuitive for the user and efficiently implemented by a computer. But a system design must also handle cases that are not so simple or intuitive. CAS problem-solving sessions abound in cases where the name and its value(s) in some context(s) must coexist. Sometimes values are not the only relevant attributes of a name: there may be a declaration of "type" or other auxiliary information. For example it might evaluate $\sin^2 x \leq 1$ to "true" knowing only that x is of type Real. (If x were known to be complex, this could be false.)

CAS builders, either by tradition or specific intent often impose two criteria on their systems intended for use by a "general" audience. Unfortunately, the two criteria tend to conflict.

1. The notation and semantics of the CAS should correspond closely to "common intuitive usage" in mathematics.

2. The notation and semantics of the CAS should be suitable for algorithmic programming, as well as (several levels) of description of mathematical objects, ranging from the abstract to the relatively concrete data representations of a computer system.

The need for this first requirement (intuitiveness) is rarely argued. If programs are going to be helpful to human users in a mathematical context they must use an appropriate common language. Unfortunately, a careful examination of common usage shows the semantics and notion of mathematics as commonly written is often ambiguous or context dependent. The lack of precision in such mathematics (or alternatively, the dependence of the semantics of mathematical notation on context) is far more prevalent than one might believe. While mathematics allegedly relies on rigor and formality, a formal "automaton" reading the mathematical literature would need to accumulate substantial context or else suffer greatly from the substantial abuse of notation that is, for the most part, totally accepted and even unnoticed by human readers. Consider $\cos(n+1)x \sin nx$, as appears in a well-known table of integrals (formula 1.351 [8]). Try that in your favorite CAS parser!

Because the process of evaluation must make explicit the binding between notation and semantics, the design of the evaluation program must consider these issues centrally. Furthermore, evaluation typically is intertwined with "simplification" of results. Here again there is no entirely satisfactory resolution in the symbolic computation programs or literature as to what the "simplest" form of an expression means.

As for the second requirement, the need for programming and data description facilities follows from the simple fact that computer algebra systems are usually "open-ended." It is not possible to build-in a command to anticipate each and every user requirement. Therefore, except for a few simple (or very specific, application-oriented) systems, each CAS provides a language for the user to program algorithms

and to convey more detailed specifications of operations of commands. This language must provide a bridge for a computer algebra system user to deal with the notations and semantics of programming as well as mathematics[1]. Often this means including constructions which *look like mathematics but have different meanings*. Examples: $x = x + 1$ could be a programming language assignment statement, or an apparently absurd assertion of equality. Furthermore, the programming language must make distinctions between forms of expressions when mathematicians normally do not make such distinctions. As an example, the language must deal with the apparently *equal* but not *identical* $2x$ and $x+x$.

Programming languages also may have notations of "storage locations" that do not correspond simply to mathematical notations. Changing the meaning (or value) of an expression by a side effect is possible in most systems, and this is rather difficult to explain without recourse to notions like "indirection" and how data is stored. For example, Maple and Macsyma provide for assignment into the compound data structure representing a matrix. An assignment may change not only the expression as expected, but some other use of the structure. Using Macsyma syntax, `a:matrix([1]); b:a;` establishes both `a` and `b` as a 1 by 1 matrix with entry 1. Then `b[1,1]:0` changes the value of `a` as well as `b`. Another somewhat related question is how one would deal with evaluation in a matrix that is really a spreadsheet of formulas and values[1].

Traditional numerical programming languages do not have to deal with the gradations of levels of representation in a computer algebra system. In fact, the common programming term "function" for "subroutine returning a value" indicates the depth of mismatch between the mind-sets common in traditional programming and mathematics respectively. What sense does it make to ask if a Fortran "function" is "continuous"? At best, Fortran functions are maps from discrete sets to discrete sets, and that alone makes the concept of continuity inapplicable. Observe that any (non-constant) "smooth" function ends up being discontinuous when you look closely. A system that can compute the derivative of $f(x)$ as (literally) something like $df(x)/dx$ must have a quite different set of notations and semantics in dealing with "functions" from those of a Fortran compiler. Furthermore, no computer algebra system today deals constructively with a statement beginning "Let C be the set of continuous *functions* ...."

In more "complete" programming languages in which symbols and values can coexist, additional issues arise. The oldest higher-level language providing conundrums about evaluation is Lisp, in which it makes sense to ask the question, when does `(eval 'x)` differ from `x` ?

In this paper we take most, but not all, of our examples from widely-distributed CAS: Macsyma, Mathematica and Maple. AXIOM, a more recently released computer algebra system with a mathematical type system, introduces another somewhat orthogonal perspective on evaluation, requiring the results to conform to a calculus of types. We believe the options presented in these systems reasonably cover the range of models that have been implemented in other current computer algebra systems. There are also programming languages which deal effectively with "symbols" including most notably, Lisp. Space does not permit discussing such programming languages generally.

With respect to its evaluation strategy, each existing system chooses its own perhaps twisting pathway, taking large and small sometimes controversial stands on different issues, along the way. It is an understandable temptation in design, to place enormous complexity in this one operation, `eval`. After establishing or implying some context, virtually all problem-solving questions can then be reduced to `eval(solve(problem))`.

As suggested above, we believe some design issues are matters of opinion, and undoubtedly some of our criticisms may irk designers who insist "I meant to do that" or "The alternatives are worse." Yes, we do not always provide a constructive and consistent solution to the problems we point out.

2 Context of Evaluation

In a typical CAS, an internal evaluation program (`eval` for short), plays a key role in controlling the behavior of system. Even though this program may not be explicitly available for the user to call, it is implicit in much that goes on. Typically `eval` takes as input the representation of the user commands, program directives, and other "instructions" and combines them with the "state" of the system to provide a result, and sometimes a change in the "state." In other words, `eval` stands between the input-parser and the operational routines for mathematical manipulation (programs to factor polynomials, integrate rational expressions, etc.) Some of the operational routines must themselves use `eval` internally to achieve their objectives. For example, evaluation of a boolean expression will generally determine the value of a conditional expression. Because of this kind of dependence, changes to the evaluation process can affect systems in many ways that do not seem to explicitly "call" `eval`.

A more careful consideration of `eval` as one might expect from a more formal programming language design looks a bit different. Not being confused by the CAS attempt to intuit the proper behavior of names, indeterminates, and values, the scope and extent of variables can be more easily determined, and some choices made available to the programmer: it should be possible to deal with contexts of binding in a computationally consistent fashion. In a flexible programming language, different bindings might co-exist within some global framework.

For example, Common Lisp has the potential for any number of distinct simultaneous bindings of a name, and also for different scoping rules regarding those names. The preferred form is lexical scope (but dynamic scope can be used by choice). Its resolution of the result of `eval`, and the discussion has, after a few decades, become rather well-formed. Consider our earlier example of what the value of `(eval 'x)` should be in Lisp.

```
(let ((x 1))   ;<--- the first x is bound to 1
  (let ((xPlusOne '(+ x 1)))
    ...
    (let ((x 2)) ; another x
      ...
      (eval xPlusOne))))
```

Is this supposed to return 2 or 3? By `xPlusOne` do we mean "find the value of the innermost `x` and add one to it"

[1] Several systems use a different system implementation language (C or Lisp, typically), where there is a clear distinction between mathematical expressions which are treated as *data* and programs. For various reasons it is generally believed that these languages are less intuitive to the mathematician, although their presence solves the programmability issue decisively for persons willing and able to use them!

or do we mean "find the value of *this* `x`, the one that's right there above `xPlusOne`, and add one to *it*? So far as intuitions go, it seems a valid intuition to think the programmer meant "the first `x`." Over the years the Lisp community has considered different ways of making `eval` work and has decided that some ways are better than others, and in the better ways `(eval 'x)` is not necessarily equivalent to `x` as an utterance[2].

3 Routes to Evaluation

There are several approaches to programming an evaluation algorithm. We describe major variants in the subsections below.

3.1 Eval, a procedure

The traditional procedural technique looks like the Lisp evaluation mechanism. The resemblance is not accidental. There is a strong similarity between an algebraic expression represented as a tree (data), and a Lisp program to evaluate that expression. For example, `(+ x (* y z))` could be either. This leads to evaluation that looks like induction:

- If an expression is an indivisible "atomic" node such as a number or a symbolic name, follow the rules for evaluating atoms. Typically these include
  - numbers evaluate to themselves (in some cases some type conversion may be done).
  - well-known constants like π may (in certain circumstances) be changed to approximate numbers.
  - A symbol (variable, indeterminate) like x may be replaced by its value, if it has one. If the value of x is another symbol or a composite expression, some evaluation schemes will evaluate that expression, perhaps until it no longer changes.
- If an expression is a composite object like $F(a, b, c)$, it has a "main operator" F. Some procedure associated with the name F, say `fproc`, will eventually be executed, either on its raw arguments, or on the result of evaluating its arguments. The typical case is that first `eval` will be applied to each of (a, b, c), and then the `fproc` will be applied. We can think of this as two steps: "evaluate the arguments of the main operator" followed by "apply the operator to the evaluated arguments". There are many variations to this technique, but in general it involves recursively evaluating subexpressions.

There is a rather substantial shortcoming in this model. Mathematical semantics cannot always be modeled adequately by this strictly bottom-up tree-traversal evaluation process. Consider the command `subs(x=a,x-x)` by which we mean to have the system substitute `a` for `x` when it appears in the expression `x - x`. A bottom-up evaluation scheme would evaluate `x - x` to 0 prior to performing the substitution, and therefore the substitution would always result in 0. But what if the computer system allows for such objects as ∞ where $\infty - \infty$ is arguably different from 0? What if `a` were an interval such as $[0, 1]$ in which case `a - a` is arguably the non-zero interval $[-1, 1]$. What if `a = O(n)`, an asymptotic order? What if the object `a` does not even admit of a subtraction operation, since it could be, for example, such a non-algebraic object as a "file descriptor"?

[2] In this paragraph I've taken some words from a note to comp.lang.lispby jeff@festival.ed.ac.uk (J W Dalton) October 18, 1993

3.2 A multi-phase model

An alternative approach is to change evaluation to a two-phase operation. The first phase, generally operating top-down, provides a context and perhaps an expected type for a result, for every operation, and the second phase, operating bottom up, computes values.

Consider the tree expression (in a lisp-like notation) of `(+ (modulo 5 3) (* 2 2))`. The root node `+` has two descendants. In traversing the tree downward, the first phase of evaluation imposes the constraint that the types of its arguments must (at least) be objects that can be added. The constraints imposed by `modulo` on its arguments as well as its result types provide further context. Perhaps in this case we intend that `(modulo 5 3)` return "-1 in the finite field $\mathbf{Z}_3$". Then note that the context of `(modulo 5 3)` affects the constraint on `(* 2 2)`, because we now presume that the `+` is addition over a finite field. Thus this first phase is not strictly one-pass; if the expression had been `(+ (* 2 2) (modulo 5 3))`, the new information passed upward to the `+` from the second argument would have to be transmitted back to the first argument. Indeed, one can envision an order of evaluation to types where the result type of an expression deeply nested in a tree can force re-evaluation of the whole tree; this re-evaluation may actually occur more than once (and in a poorly constructed coercion system, might cycle without convergence).

In the second phase, the relations between arguments and results of each of the operators could be examined to determine the actual computation to be performed. In some cases this might still be ambiguous because the type of an expression might depend on values to be computed. For example, in the expression `(if b 1 1.0)`, if the (presumably boolean) value for `b` is true, the result is a (presumed) integer, but otherwise a (presumed) floating-point number. Perhaps such expressions should be forbidden, or their results should be forcibly typed to the union of their possible types. Later, it may be necessary to "retract" the type to one of the constituents.

The AXIOM system has made a case that the first phase can be done on input in an interpretive mode, and that after that point, every object has a type; every operation has a set of coercions for any n-tuple of argument types.

Although this is very helpful, it is painfully unclear, even in some relatively simple cases, how the second phase should coerce types when they do not match. We will return to this in a later section.

3.3 Evaluation by Rules

Another technique, not particularly matched to the mathematical context of computer algebra systems, but nevertheless plausible because it can be used to define general algorithms on trees, is to define transformations by rules.

These rules might direct transformations like

$$\forall x, y \quad \log(x \cdot y) \to \log(x) + \log(y) \quad x, y > 0$$

or

$$|x| \to \begin{cases} x, & \text{if } x \geq 0; \\ -x, & \text{if } x < 0; \\ |x|, & \text{otherwise.} \end{cases}$$

Since even highly constrained and very simplistic rule transformation systems (e.g. Post systems [4]) are equivalent to Turing machines in formal computational power, they could be used, in principle, as the sole computational description for any formal algorithm, not just evaluation. Their advantage over the more traditional procedural definitions seems to be primarily in their declarative nature. They may be simpler for a "user" of a complex system to understand. An approach adopted by Theorist, a highly-interactive commercial CAS, seems especially worthwhile: it can be set up to present transformations from a list of rules selected from a menu of rules. This seems much more palatable than either writing rules from scratch or trying to figure out how some system's "expand" command will change an expression from the documentation, test cases, and guesses. In brief: the rules also serve as documentation.

There are counterbalancing disadvantages to using rules:

1. It is possible to define rules so that more than one rule can apply. In fact, it is often difficult to write a complex transformation without overlapping rules. A simple conflict resolution process such as "the most recently defined rule takes precedence" may lead to very slow execution—essentially all rules must be attempted to make sure that none can be applied. Also, it might not provide the expected result. Another resolution such as "the most specific rule takes precedence" can be difficult for the human "programmer" to understand or for the computer system to apply. (In fact this meta-rule is non-computable, generally.)

2. Much of the software technology of modularity, functional programming, information-hiding, etc. is either irrelevant or takes a rather different form in terms of rules. It is rarely convenient to use a completely declarative approach to specify a large computation. Early evidence from the expert-system building community suggests that constructing large systems by programming rules [2] may be even more difficult than construction via traditional imperative programming.

3. The unit of specification — rule application, is really a two part process. It requires pattern matching (usually a kind of graph-matching) in conjunction with checking of predicates on pattern-match variables. The second part is substitution and evaluation.

   The matching process can be very costly, even if it results in a failure to match. It may be important to develop heuristics which avoiding the attempt to match at all. Such attempts to "optimize" rule sets, where information about partial matches are propagated can be critical to speed. There is a literature on computing reductions related to rules (e.g. Gröbner, Knuth-Bendix [3]), which can be helpful in the domain of polynomial computer algebra evaluation problems have to do with computing in polynomial ideals—reduction of systems modulo polynomial side-relations in several variables. Unfortunately, success in a larger domain does not follow from this same theory.

4. Matching trees where subtrees are unordered is inherently exponential in the depth of the trees. Expression or pattern trees with root nodes denoted "plus" or "times" have unordered subtrees. If such commutative matching were an inherent part of the evaluation process, this would not be a disadvantage of rules versus other mechanisms; however, some costs in evaluation via commutative tree searches seem to be more an artifact of the mechanism of rules than a requirement for evaluation.

3.4 Object-oriented Evaluation

An object-oriented approach to evaluation provides another perspective. Each object, an expression represented (abstractly at least) as a tree, has a root or lead operator. This is associated with a program that "evaluates" objects of that type. Thus one would have a "plus" evaluator, a "times" evaluator, etc. An elaboration on this idea would provide for inheritance of other information: The "plus" evaluator might inherit routines that were associated with the tree-nodes (objects being added) which might, for example, be members of the ring of integers, or power-series in some variable. Objects with no evaluation functions (e.g. a newly introduced $f(x)$), could also inherit some default evaluation mechanism from the "mother of all evaluation routines." Such a default routine might return $f(y)$ where y is the evaluated form of x.

An object-oriented programming approach is a handy way to organize programs along orthogonal lines to correspond to helpful conventions from mathematics and data structures.

4 Common Problems

4.1 Failure of the Top-Down Model

Each of the evaluation models generally boils down to a descent through an expression tree, reserving operations while evaluating operands, and then backing out. Let us review the results of a sequence of computations in evaluating the expression $f(g(x,y),h(z))$. Here the evaluator acts on f, g, x, y, and then applies g to (the evaluation of) x and y. Then h, and z, are visited and h is applied to (the evaluation of) z. Finally f is applied to $g(x,y)$ and $h(z)$.

This sequence is sometimes wrong, because it assumes that the evaluation of $g(x,y)$ is independent of (say) z. How might this not be the case? As a somewhat frivolous example mentioned earlier, consider $(a+b)+c$ where $a=5$, $b=6$ and $c=2 \bmod 5$. After adding a and b to get 11, we then discover that arithmetic should have been performed modulo 5. A less frivolous example along the same lines would be one in which (say) a, b, and c are power series in which arithmetic combining a and b must be redone in order to combine the result with c. Yet another example (using Mathematica syntax) is `N[Integrate[ ...]]` where the `N` means "numerically". If we first evaluate the argument, then the symbolic integration will be attempted, rather than a numerical quadrature.

Another consideration that is sometimes at the core of performance improvements is whether "evaluation" and "simplification" should be interleaved. This can be illustrated by the famously inefficient Fibonacci number recursion:

$$f(x) \to \begin{cases} 1 & \text{if } x < 2; \\ f(x-1)+f(x-2) & \text{otherwise.} \end{cases}$$

We can use the sequence $f(4) \to f(3)+f(2) \to (f(2)+f(1))+f(2) \to (f(1)+f(0))+f(1)+f(2) \to \cdots$ or $f(4) \to f(3)+f(2) \to (f(2)+f(1))+f(2)) \to [\mathbf{simplify}] \to 2f(2)+f(1) \to \cdots$

The latter sequence of operations is much faster, since it cuts down the exponential nature of the recursion (not that it is efficient either.)

For systems which use many different data types or allow parameters in the data types (an example of an implicit parameter is the matrix dimension in a type "square matrix" or the list of variables in a multivariate polynomial, or the coefficient domain for polynomials), some form of non-local information may be required to determine a type for the result.

In Macsyma, for example, converting an expression to a polynomial form requires two stages. First is a linear-time scan so that all the symbol-names in the expression are known (so that the "main" variable and the others can be presented in sorted order[3]) It also notices some simple relationships like z^{2n} being the square of z^n. The second stage then combines the given expressions conceptually, in the appropriately-constructed domain of rational functions extended by the set of variables or kernels found.

If this conversion is not done initially by a two-pass algorithm, the evaluator may end up "backing and filling" re-representing sub-parts, and especially non-obvious kernels.

Another somewhat different problem that may not appear to depend on types or domains, but arguably does, has been mentioned previously: The short-cut which replaces with 0 any expression that looks like $x - x$ is not always valid unless the domain of x is known. If x is a stand-in for ∞, should the substitution be made? (Perhaps yes, arguing that both ∞ symbols are shorthands for a single variable.)

4.2 Quotation, Nouns and Inert Functions

In a computer algebra system it is useful at least on occasion to deal with temporarily "unevaluated" objects, even though they might be evaluated in the current context to yield something else. Consider typing a differential equation into a system. One might wish to type `diff(y,t)=f(t)`. But then if a "normal" imperative interpretation of `diff` were applied, `diff(y,t)` might very well be evaluated to 0: y does not apparently depend on t. As another example, consider a program that uses, as intermediate expressions, the symbolic roots of a quartic equation. This might happen when a computer algebra system expresses the answer to certain classes of (elliptic) integrals involving rational functions of square-roots of quartics. It makes much better sense (and saves considerable time and space) to approach such problems by first abbreviating the roots by making up names, say $\{r_1, r_2, r_3, r_4\}$, and then expressing the answer only in terms of these roots. "Full evaluation" would ordinarily dictate that if you have an expression for r_i then you are compelled to eliminate r_i from expressions in which it occurs. In the case of quartic roots, this is quite hazardous, since the roots can each take a page to typeset, are pretty much guaranteed not to simplify much in isolation, and yet combine with each other rather in neat ways that traditional simplification routines will miss. Unless the roots are in fact quite small (such as the case of all floating-point approximations — not symbolic at all) or one can apply special simplifiers to collapse expressions involving subsets of $\{r_1, r_2, r_3, r_4\}$, it is probably best to leave the answer in terms of those roots.

Consider also the plotting of a function
`f(x) := if (x>0) then x else -x`.
If the command is `plot(f(t),t,-10,10)` or something similar, one must evaluate the first argument just one level: from `f` to its definition, but no further evaluation of the definition is possible. If one foolishly answers the question "Is $t > 0$?" with "no, t is just a symbol" then you have lost. One must defer even asking this question until the `plot` program repeatedly evaluates the expression for different values of t.

In Lisp, such issues are dealt with directly. There is a "quote" operator (generally with the notation `'x` meaning "the unevaluated symbol `x`") to ward off the effect of the general evaluator. In Macsyma, a similar notation for quoting operations is available so one can write a differential equation as `'diff(y,t)+f(t) = 0`. For the quartic equation problem one could let `s=solve(...)` and then deal with `'s[1]` etc. without looking at the form of the solution.

A hazard here is that one does not want to see—displayed—the quote-marks, suggesting the need for a slightly different but more visually similar "noun" operator for `diff`. Maple calls such operators "inert" and uses a first-capital-letter convention to distinguish them from the normal "verb" operators. The Maple convention is to identify such operators with their lower-case versions only in a particular evaluation context where such inert operators can be removed by a special evaluation scheme. For example, the normal evaluator will remove (by evaluation) derivatives (the `diff` operation) but will leave `Diff` unchanged. Using an inert integration operation in an expression leaves the form untouched until some subsequent evaluator (say one for numerical solution of differential equations) treats the inert operator in some special way.

Although Macsyma has a mechanism for forcing its evaluator to convert a particular "noun" form to a "verb" form, this is not quite analogous to Maple's behavior, which seems to generally take the view that a global re-examination of the expression is needed to remove inert operators. (In Lisp, the function `eval` "undoes" the quote.) There are subtle issues as to how to resolve the bindings of symbols that are contained in the previously-inert expression being evaluated. Various careful definitions can be seen in the Common Lisp and Scheme standards; most computer algebra system documentation seems to ignore the issue in the hopes that the user will not notice the vagueness at all.

An example of another inert function in Maple may help clarify the concept. The operation `Power` is used in conjunction with the `mod` operator to provide "special evaluation" facilities: to compute $i^n \bmod m$ where i is an integer it is undesirable to compute the powering first over the integers (possibly resulting in a very large integer) before reduction modulo m. The expression `Power(a, b) mod p` which may also be written as `a&^b mod p` is similar in form but it constructs an expression "inertly" without evaluation, and then afterward in the "mod p" context, computes the power, avoiding integers larger than m. Another example of an inert function is `Int`, short for integrate. This inert function can be removed by numerical evaluation in `evalf`, and its use is particularly time-saving when the user can predict that symbolic integration will not result in a closed form (and therefore should not even be attempted.) It may of course result in wasting time when a symbolic result is easy to compute and then evaluate numerically. Maple does have a function `value()` to change a given inert form to an active form, on request.

4.3 Confusing Arrays and Matrices

An array is a data structure for storing a collection of values indexed by some set. The set is usually a range of integers, pairs of integers, or n-tuples of integers. One might consider the use of other index sets such as letters or colors. Although it is convenient to have an ordered index set, it may not be required. Operations on arrays include access and setting

[3] By contrast, Maple does not sort its variables. (They are "ordered" by the accidents of memory location).

of individual values; occasionally accessing and setting of sub-arrays (rows, columns, blocks) is provided. Sometimes extension by a row or column (etc.) is possible.

Naively, a matrix appears to be simply a case of an array with an index set of full rectangular or square dimensions. (Along with most computer algebra systems we will ignore the very important efficiency considerations that accrue to special forms of matrices: diagonal, block diagonal, triangular, sparse.)

However, the operations on matrices are quite different from arrays. For matrices of compatible sizes and entries, one can compute $A := B^{-1}A$. Issues of "where does one store B^{-1}" don't occur. Nor does the user have to worry about the storage of entries in A on the left messing up the entries in A on the right. Copying over data is done automatically. In a classical numerical language, one would probably have to allocate array space for one or two intermediate results.

Evaluation of a matrix is a problem: If one asks for $A_{1,1}$ does one evaluate the expression to the entry, or does one evaluate that entry? In a system that does "evaluate until nothing more changes" this may not matter; in a system that evaluates "once" does the access to an element count as that one evaluation? Is one allowed to change one element of a matrix, or must one re-copy it with a changed entry? It may make sense to forbid altering a matrix after it is created. In Mathematica there seems to be the additional (usually unexpected and ill-advised) possibility that the symbol A may have rules associated with it requiring re-evaluation of the whole matrix A when only one element is accessed.

Then there are issues of subscripted symbols. If nothing other than the name A of a matrix or array is given, is reference to $A_{3,3}$ an error ("uninitialized array element", "[potentially] out-of-range index for array") or simply the unevaluated $A_{3,3}$? Mathematica makes no distinction between a subscripted name and a function call (`A[3,3]` or `Sin[x]`). They are both "patterns" subject to replacement by rules and evaluation.

And sometimes it is important to deal with the *name* of an array, even if its elements are explicitly known. So-called implicit operations can be very useful, and it is valuable to be able to simplify AA^{-1} to I knowing only that A is a non-singular square matrix, and not referring to its elements at all. Indeed, not even knowing its size.

5 Infinite Evaluation, Fixed Points, Memo Functions

So-called infinite or fixed-point evaluation is attractive primarily because it is commonly confused with simplification. The requirement is to detect that any further application of a simplification program "won't matter" —that the expression or system has reached a stable state or a fixed point, and further attempts to simplify the expression will have no effect. Thus if you were to re-evaluate infinitely many times, it would not change.

Let us define simplification for our purposes here as a transformation of an explicit function $f(x)$ in some parameter x or vector of parameters, to another explicit function $g(x) = \mathtt{simp}(f(x))$ such that for any valuation v given to x *in some agreed upon domain*, $f(v) = g(v)$ and moreover, $g(x)$ is by some measure less complex. For example $f(x) = x - x$ and $g(x) = 0$ are a plausible pair: g has no occurrence of x and hence might be considered simpler. This equivalence is however false if the domain of valuation is that of interval arithmetic: if $v = [-1, 1]$ then $v - v$ is the interval $[-2, 2]$, not 0.

A very appealing and generally achievable attribute of a good simplification program is idempotence. That is, `simp(`x`)=simp(simp(`x`))` for all symbolic expressions x. It is intuitively appealing because if something is "already simplified" then it cannot "hurt" to try simplifying it again. Since simplification *per se* should not change the environment, it is plausible that a valid simplification routine applied repeatedly will not cycle among some set of equivalent expressions, but settle on one, the "simplest". (This is not to say that all equivalent expressions will be simplified to the same expression. Though that would be desirable (a Church-Rosser [4] simplifier), for some classes of expressions, it just happens to be undecidable.) Note that we could consider building a valid simplifier by defining a sub-simplification procedure which is applied repeatedly until no more changes are observed, and then this n-iterative process is the `simp` with the idempotence property.

Some aspects of evaluation are almost indistinguishable from simplification, especially if the valuations v are chosen from "expressions" in the same domain as f. Repeatedly associating valuations v with their names x leads to problems. Infinite evaluation can work only if `eval(eval(x)) = eval(x).`

Unfortunately, if the usual assignment statement `x:=x+1` is treated in this manner, and the "value" is nominally the right-hand side of the expression, there is no finite valid interpretation.

But if finite evaluation must be used in that situation, how is one to determine "how many times" to apply a rule such as $ax + bx \to (a + b)x$? Consider $3x + 4x + 5x$. Can one application do the job of fully evaluating the result of applying the rule?

There are actually arguments that it can. Application of a rule, or more generally, rule sets, can be sequenced in a number of established ways, although termination is difficult (theoretically impossible in some cases) to determine. See the Appendix on rule ordering for further discussion of this point.

If an expression is always simplified or evaluated to a particular form, why not remember the input-output relationship and short-cut any attempt to *repeat* the calculation by referring to the "oracular" evaluator? Indeed, one of the principal efficiency tricks made available to programmers in any of the systems is the notion of a "memo function". In Macsyma, so-called hash-arrays are used for this, Mathematica has a comparable facility by means of its rule-based memory, and the Maple programmer inserts `option remember` in a procedure definition to use this facility.

By using these facilities, any time a function is "called" on a given set of arguments (in Macsyma or Maple it looks more like an array reference), the set is looked up. If it is a new set, the result is computed and then "remembered" typically in a hash table with the argument set as an index. The second and subsequent times the result will be remembered from the first time, and simply recalled. This can be a potentially enormous improvement, but it has the unhappy consequence that if "impure functions" (that is, procedures that have side-effects, or whose results depend on global variables) are used, the remembered results may be inappropriate. Thus access to a global variable in Maple `g:=proc(z) option remember; z+glob end;` refers to the global variable `glob`. If `glob=1` then `g(3)` will be 4. Changing `glob` to 2 does not make `g(3)` be 5. It is "stuck" at 4.

Functions having histories are not necessarily restricted

to *user-defined* programs. Maple system programs are set up with `option remember`, including `factor`, `normal`, `simplify` and (at one time) `evalf`. Some subtle problems reported as Maple bugs are caused by such memory. For example, re-computing a function after setting the system `Digits` to compute with increased numerical precision might appear to have no affect: the earlier low-precision result may simply be recalled from memory and new values not recomputed at all.

The negative consequences of this are quite far-reaching in all of the systems and can be most unfortunate: fixing a program may not repair an incorrect answer because of an entry in the memory table of some function `f`, whose name may not even been known to the programmer. Such memory is cleared by Maple's `forget(f)`, Mathematica's `Clear[f]` or `Remove[f]` or Macsyma's `kill(f)`.

Users and novice programmers can easily misunderstand what has happened, resulting in substantial debugging difficulty. I suspect that experienced programmers fall prey to this source of bugs as well, especially since they may be more inclined to try to take advantage of the vast speedup potential.

6 A Collection of Systems

6.1 AXIOM

Computing the "value of an expression e" in AXIOM [10] resembles the notion of evaluation in Lisp, which is to say, it is evaluation with respect to an environment. It also has an additional component, which requires evaluation to a type.

Let us give several simple examples. Consider $p = 3x+4$, a polynomial in $\mathbf{Z}[x]$, the ring of polynomials in the indeterminate x over the ring of integers $\mathbf{Z}$. What is $p/3$? Plausibly it is $(3x + 4)/3$, an element in the quotient field $\mathbf{Z}(x)$, namely a ratio of polynomials in $\mathbf{Z}[x]$ (In AXIOM this is `type: Fraction Polynomial Integer`). Alternatively and perhaps just as plausibly, $p/3$ is $x + 4/3$, an element in the ring $\mathbf{Q}[x]$, namely a polynomial in the indeterminate x with coefficients in the field of rational numbers, $\mathbf{Q}$. This is AXIOM `type: Polynomial Fraction Integer`. Since there is only one intuitive conventional notation for division covering both cases, one solution, and perhaps the wrong one for future computation, will be chosen in any situation where the result must be deduced from the symbols p, /, and 3. Conversions are possible, but there are intellectual and computational costs in using the wrong form.

A slightly more complicated, but extremely common, design situation occurs when performing arithmetic in $\mathbf{Z}(x, y)$ in preparation for rational function integration. A computer algebra system would like to deal with the "correct" form: if one is integrating with respect to x this is to coerce the expression to a ratio of polynomials n/d where n and d are each in $\mathbf{Q}(y)[x]$ and d is monic (has leading coefficient 1). This is quite asymmetric with respect to order of variables: integration with respect to y would require a different form, and integration of d/n may look quite different from a simple interchange of numerator and denominator from n/d. As a simple instance of this, consider the expression $(1 + y)/(3 + 3xy)$. Integration of this expression with respect to x is particularly trivial if it is first rewritten as $(1/3) \cdot (1 + y)/y \cdot (1/(x + 1/y))$ The integral is then $(1/3) \cdot (1 + y)/y \cdot \log(x + 1/y)$

AXIOM goes further than other widely available systems in making the descriptions of such domains plausible. In attempting to provide the tools to the user to construct various alternatives, it does not necessarily provide the best intuitive setting. For example, embraced within the notion of polynomial in several variables are the categories of `Polynomial`, `Multivariate Polynomial`, `Distributed Multivariate Polynomial`, `Sparse Multivariate Polynomial`, `Polynomial Ring` and others. These domains are not necessarily distinguishable mathematically, but in terms of data handling convenience. Their distinguishing *efficiency* characteristics may not be meaningful to a user who is mathematically sophisticated but inexperienced in computer algebra. While it may be comforting to some to have a solid algebraic basis for all computations, the user without a matching algebra background may encounter difficulties in formulating commands, interpreting messages, or writing programs.

A subtlety that is present in all systems but perhaps more explicit in AXIOM is that one must also make a distinction between the types of variables and the types of values. For example, one could assert that the variables n and m can only assume integer values, in which case `(+ n m)` is apparently an integer. But it is manifestly not an integer as we have written it, and as long as n and m are indeterminates, the sum of the two is an expression tree, not an integer.

Given that we have a model in AXIOM that provides a kind of dual evaluation: evaluation of an expression to a pair: (type, value), how does it work? It appears that by converting the type, one achieves much of the pseudo-evaluative transformations. Thus if we are given `r := 1/3` `(type: Fraction Integer)` then the command `r :: Float` results in `0.3333333...` where the number of digits is set by the `digits` function. Of course this is not an exact conversion of value—more than the type is altered.

The principal other kind of evaluation in AXIOM is simple:

Any occurrence of a name in an expression in a context to be "evaluated" such as the right-hand-side of an assignment, or an application of a function, causes the current binding of a name to be used in place of its name. That is `p := 1/3` establishes a value for the current binding-place for `p`. References to `p` in the ordinary course of events will provide `1/3`. A single quote `'p` prevents such evaluation. This is essentially the Lisp model, except that the values used may themselves have names, and these too are "evaluated" potentially infinitely.

There is one form of `eval` that "removes" all quotes—in its sole argument. Another set of evaluation transformations are based on substitution semantics: that is, they specify the substitution of some value(s) for a symbol or name(s) in an expression (where then name could be a variable or an operator).

Perhaps confusingly, syntactically indistinguishable versions of `eval` include operations on symmetric polynomials, permutation groups, and presumably anything an AXIOM program or a user wishes, as long as they can be distinguished by the types of the arguments. The expression `evaluate(op)` identifies the attached function of the operator `op`. Attaching an `%eval` function `f` to an operator `op` is done by `evaluate(op,f)`. The Common Lisp convention `(setf (evaluate op) f)` might be clearer way of indicating this.

R.D. Jenks of the AXIOM group at IBM has kindly provided additional details.

A mapping from e to its value $V(e)$ looks like this.

If e is a literal symbol, say `x` then $V(e)$ depends on how `x`'s binding in the current context was assigned. If it was a "`:=`" assignment (`x :=`a) where $V(a)$ was y at the time,

then it is y. If it was a "==" binding, (`x ==`e), then $V(x)$ is $V(e)$. If there was no assignment, then it is an object of type Symbol, `x`.

If e is a compound expression, then it has an operator and operands. It also has a context in which a type u is expected for $V(e)$. To evaluate e of the form $f(a_1, \cdots a_n)$ in the compiled language,

1. Let $A_i = V(a_i)$ for $1 \leq i \leq n$.
2. Check to see if there is a unique "signature" denoted $f : (B_1, \cdots, B_n) \rightarrow B$ in the environment such that for each i, $1 \leq i \leq n$, A_i is a subtype of B_i and such that B is a subtype of type u. If so, apply that operation to produce the value of $V(e)$

The semantics of the interpreter differ from the compiled code in that one replaces the notion of "is a subtype of" with "can be coerced to" and in case more than one signature is found, choose the one judged to be "of least cost".

Exceptions to the general scheme are needed for a few special operators.

For example: When `x` is a variable, `x := e` assigns a value (the equivalent in Lisp is `(setq x e)`); `f(a) == e` or `f == (a) +-> e` defines a function approximately like `(setq f '(lambda(a) e))`.

A detailed examination of evaluation in $A^\sharp$, (the language underlying Axiom) is beyond the scope of this paper. In some circumstances the run-time computation should be unimpeded by considerations of type, but in others it can involve a good deal of machinery. Functions, domains, and categories are first-class objects, and appropriate coercions are sometimes required.

6.2 Macsyma

To a first approximation, Macsyma evaluates everything once, as in Lisp. Just as in Lisp, there are special forms that don't evaluate all their "arguments" (assignment operators don't evaluate their left-hand operands); there is also a form analogous to Lisp's `eval` (namely, `ev`) that evaluates one extra time. And there is a quote operator (the prefix apostrophe) which one thinks of intuitively as a way to prevent an evaluation. Actually, the evaluation happens, it is just that `'x` evaluates to `x`.

Contrary to Lisp's convention, evaluating a symbol `x` that has no value does not result in an error, but merely returns the symbol `x`. It is as though the system figured out that the user meant to type (or was too lazy to type) `'x` when the evaluation of `x` would otherwise signal an error of type "unbound variable". An attempt to apply an undefined function to arguments would ordinarily signal an error of type "undefined function" but here merely constructs a kind of quoted "call".

Experimentation with such language decisions is fairly simple. In fact, one can easily provide a simple alternative to the Lisp evaluator, written in Lisp, but using these rules. Such a program is given in Appendix III. This model is appropriate for the "functional programming" approach where the value of a function depends only on its arguments and not on its context. More work might be needed to provide a natural way of expressing contexts (via an environment passed downward). Such an environment would be used to distinguish between the evaluation of `(^ 3 5000)` and `(mod(^ 3 5000) 7)`. In the first case, the `^` would mean computing the 5000 power of 3 over the integers; in the second, the powering algorithm should be a much faster "mod 7" version.

There is are option for "infinite evaluation" in which case an expression is evaluated until it ceases to change. This can be done by a "command" `INFEVAL` or set up in an environment by using `ev(...,infeval);`. A related procedure is `INFAPPLY`, which takes a function and arguments.

Evaluation and Simplification are two intertwined processes: Commands that are submitted to the system by a user are first evaluated — symbols' values are inserted for their names, functions applied to arguments, etc. Next, the simplification program makes a pass over the answer, in many cases rearranging the form, but not the "value".

The user can change the meaning of evaluation by supplying values for symbols, function definitions, and setting some flags (for example, `numer:true` means that 1/2 becomes 0.5 and constants such as π are given floating-point values.

The user can change the meaning of simplification by advising the system of rules via `tellsimp` and `tellsimpafter` which intersperse (before or after the built-in procedure for an operator) additional transformations. The process of applying rules will ordinarily require evaluation (and simplification) of the right-hand-sides of rules. It is also possible to declare a host of properties on operators that impose rules of (for example) linearity to instruct the simplifier that $f(a+b)$ should be written as $f(a) + f(b)$ etc. It is also possible to disable the simplifier by `simp:off` which is useful when one wishes to (presumably temporarily) compute with unsimplified expressions. This can be useful in, for example, telling the simplifier that 0^0 is to be rewritten as U rather than signaling an error. This requires that 0^0 first be left unsimplified in the rule-entry process.

There are some commands which are executed during evaluation which have as their effect an extra simplification of their argument. For example, `ratsimp` is such a command.

Often the user need not know whether it is the simplifier or the evaluator that changes `sin(0)` to `0`. Advanced programming requirements sometimes lead the ambitious into having to consider `noun` and `verb` forms of operators. The `noun` idea appears in Maple as `inert` operators—placeholders that however contain reminders of what they might mean if converted to verbs. Integral and differential equations typically use `noun` forms as such placeholders.

Macsyma has several different alternative evaluation (actually, simplification) schemes for special classes of representation. There is a "contagious" polynomial or rational form which can be initiated by forcing some component of an expression into this form: e.g. `x:rat(x)` will do so. In this case rational functions (ratios of multivariate polynomials over the integers in a particular recursive form) will be used as a default structure. Similar contagion affects expressions involving series and floating-point numbers.

6.3 Maple

Normal evaluation rules in Maple are "full evaluation for global variables, and one-level evaluation for local variables and parameters." That is, a Lisp-like one-level evaluation is assumed to be most appropriate for programs, and an "infinite" evaluation—keep evaluating until a fixed point is reached—in the top-level interactive "user" environment. Evaluation is accompanied by simplification always, although some special simplifications can be separately applied. There are a number of functions that don't use the standard top-down model of evaluation, but must look at their arguments

unevaluated or evaluated in a specific order. These "functions" include `eval, evalf, evaln, assigned`.

In normal Maple usage, the user is unlikely to need to use the `eval` function. There is a quote operation: `'x'` evaluates to `x`. Typically this is often used in a convention whereby a function returns extra values by assignment to quoted names. Thus `match(..., 's')` returns true or false. In case `match` returns true, it assigns a value to `s`. Used indiscriminately, this convention could lead to dreadful programs.

Maple's normal evaluation procedure can be explicitly called from within a program, for "extra" evaluation as `eval(x)`. This provides infinite evaluation as done at the top level. An optional second argument provides for multiple-level evaluations: `eval(x,1)`, which is commonly used, means evaluate variables that occur in `x` only to their immediate values, and not to continue *ad infinitum*. Because `eval` uses its second (optional) argument to control how its first argument is evaluated, the function `eval` is on the list of functions that do not evaluate their arguments.

Maple's attempt to affect simplification by imposing (say) linearity on a function is, in Maple V, mistakenly confused with function definition. Declaring an operator to be linear appears to *replace* any previous definition with one like this (from Maple V r1; newer versions have somewhat different results):

```
proc(a)
options remember;
    if type(a,constant) then a*'procname(1)'
    elif type(a,`+`) then map(procname,a)
    elif type(a,`*`) and type(op(1,a),constant) then
        op(1,a)*procname(subsop(1 = 1,a))
    else 'procname(a)'
    fi
end
```

To say that a function is both `linear`[4] and has other properties or evaluation semantics, seems beyond the scope of this "hack".

The Maple design becomes rather complicated, and seems to suffer from a surprising number of variations or alternatives to `eval` that have evolved. I suspect this has been caused by the rigid discipline imposed on the system by keeping its kernel code small and relatively unchanging over time. Thus extra pieces have been grafted on from outside the kernel, in not-necessarily-orthogonal ways.

Perhaps the most straightforward alternative to evaluation is `subs` or substitute, which is a syntactic operation—`subs(a=b,a)` is `b`. The others include `Eval, evalf, evalm, evaln, evalhf, evalb, evala, evalc, evalr, evalgf`[5].

And other issues whose import were not apparent at the design stage were inadvertently botched; these omissions sometimes become apparent only much later.

> Indeed, at this time Maple does not support nested lexical scoping. The situation may be best understood as follows "In a procedure body, each variable mentioned is either a formal parameter or local of that immediate procedure, or else it is global to the entire Maple session." [(Diane Haglund, Maple Technical Support, `sci.math.symbolic` March 31, 1994)]

[4] Maple seems not to distinguish cases such as "linear wrt x" from "linear wrt y".

[5] M. Monagan concedes that some of the functions currently in Maple should not be called eval functions, but these designations may be merely historical. Some may be eliminated (evalgf for example).

In mail to the Maple user group (April 19, 1994), including an explanation on how to simulate lexical scoping via substitution, `unapply` and quoting, M. Monagan adds, "I hope to add nested scoping rules [in Maple] soon because I'm tired of explaining this to users – I think I am getting close to 100 times!"

In spite of this plethora of programs, the ambitious programmer would not be able to take advantage of useful "evaluation-like" feature of Maple without looking at programs with names not directly identified as `eval`-something by the designers. There is `modpol(a,b,x,p)` for evaluation of `a(x)` over $Zp[x]/(b(x))$ and `e mod m` for evaluation of `e` over the integers modulo `m`.

Many additional functions are contained in the complicated suite of facilities entered by using the `convert` command. Some of the conversions are data-type conversions (say, from lists to sets), but others are form conversions such as partially factoring a polynomial, which maintain mathematical equivalence. Other uses defy simple modeling. The Maple command `subsop(0=f,g(r,s))` substitutes `f` for the 0th (operator) `g` in `g(r,s)` to return `f(r,s)`. Similarly, `convert([3,b],`*`)` returns `3*b`, but neither one of these expressions has a 0th operator: The expression $3x$ is encoded as a standard product and has no operator `*` at all.

The command `convert(10,hex)` gives the symbol `A` (which may, of course, have a value associated with it.)

The on-line manual for Maple V release 1 lists the following pre-defined conversions:

```
`+`      `*`       D        array     base      binary
confrac  decimal   degrees  diff      double    eqnlist
equality exp       expln    expsincos factorial float
fraction GAMMA     hex      horner    hostfile  hypergeom
lessthan lessequal list     listlist  ln        matrix
metric   mod2      multiset name      octal     parfrac
polar    polynom   radians  radical   rational  ratpoly
RootOf   series    set      sincos    sqrfree   tan
vector
```

The user is invited to make additional conversions known to Maple.

Why are we making such a fuss about `convert`? It is just that Maple is inconsistent with regard to what constitutes conversion, evaluation, or just a command. Why is `factor` a separate command, but square-free factoring a "conversion"?

Let us turn to those other `eval` relatives. What do they compute? The Maple manual (on-line) provides descriptions for each of them, which we quote or paraphase below.

The program `evalf` evaluates to floating-point numbers those expressions which involve constants such as πPi, e, γ, and functions such as exp, ln, sin, arctan, cosh, Γ, erf. A complete list of known constants and functions is provided.

The accuracy of the result is determined by the value of the global variable `Digits`. By default the results will be computed using 10-digit floating-point arithmetic, since the initial value of `Digits` is 10. A user can change the value of `Digits` to any positive integer. If a second parameter, n, is present the result will be computed using n-digit floating-point arithmetic.

`evalf` has an interface for evaluating user-defined constants and functions. For example, if a constant K must be evaluated by calling a procedure, then the user must define a procedure called `` `evalf/constant/K` ``. Then calls to `evalf(K)` will invoke `` `evalf/constant/K`() ``.

If `evalf` is applied to an unevaluated definite integral then numerical integration will be performed (when possible). This means that the Maple user can invoke numerical integration *without first attempting symbolic integration* through the following subterfuge. First use the inert form `Int` to express the problem, and then use `evalf` as in: `evalf(Int(f,x=a..b))`

A similar function, `evalhf` is provided that computes using hardware floating-point. Its limitation are generally those of the double-precision arithmetic system on the host computer. It will signal an error if any of the data cannot be reduced to a floating-point number. In particular, a name with no associated value will force an error.

`evala` evaluates in an algebraic number field, and `evalgf` evaluates in an algebraic extension of a finite field. These are related in that they each set up an environment in which a number of specific commands take on different meanings. For `evala` an algebraic number field is specified by the second argument. For `evalgf` a prime number is provided by the second argument. If the second argument is not provided, say, as `evala(Gcd(u,v))`, then the `GCD` function is performed in the smallest algebraic number field possible.

The commands that take into account the algebraic field include

```
Content Divide Expand Factor Gcd   Gcdex Normal
Prem    Primpart Quo  Rem Resultant Sprem Sqrfree
```

For other commands, the first argument is returned unchanged, after first checking for dependencies between the `RootOf`'s in the expression.

If a dependency is noticed between RootOf's during the computation, then an error occurs, and the dependency is indicated in the error message (this is accessible through the variable `lasterror`.)

An additional argument can be specified for Factor. This is an algebraic number, or a set of algebraic numbers, which are to be included in the field over which the factorization is to be done.

An example:

```
> evala(Factor(x^2-2), RootOf(_Z^2-2));
                      2                      2
        (x + RootOf(_Z  - 2)) (x - RootOf(_Z  - 2))
```

Note that the commands are **not** identical to those available outside the `evala` environment — they have initial capital letters and are so-called "inert" functions until they are activated by being evaluated in an environment. Interestingly, `evala(Factor(x^2-2, RootOf(_Z^2-2)));` with parentheses moved, produces the same result. (Normally an `alias` would be used to provide a name for the `RootOf` expression, dramatically simplifying the appearance of the problem and its answer.)

`evalm` evaluates an expression involving matrices. It performs any sums, products, or integer powers involving matrices, and will map functions onto matrices.

The manual notes that Maple may perform simplifications before passing the arguments to `evalm`, and these simplifications may not be valid for matrices. For example, `evalm(A^0)` will return `1`, not the identity matrix. One simple way out of this problem is to use a different operator, `^^` for matrix powers (Macsyma does this, and a later release of Maple provides `&^`).

Unassigned names will be considered either symbolic matrices or scalars depending on their use in an expression. This is probably a bad idea, and leads to strange extra notations that include `&*(A,B,C)` to multiply three matrices.

Among commercial computer algebra systems, it appears that only AXIOM has a "clean" route out of this mess by requiring that types be maintained throughout a computation. To alleviate the user from the painful chore of figuring out the types of expressions, the AXIOM interpreter heuristically infers types on input. Unfortunately, the type it infers and the type needed by the user in further steps may not agree. The clean route may thus not lead to a solution without more work.

Maple's `evalb(x)` forces, to the extent possible, evaluation of expressions involving relational operators to the Boolean values `true` or `false`. If Maple is unable to reduce the expression to one of these, it returns an unevaluated but perhaps transformed expression. For example, `a>b` will become `b-a<0`. Since Boolean operators (`and, or, not`) evaluate their arguments with `evalb`, `evalb(x)`. `(not(a>b)` is thus `not(b-a<0)`. Somewhat uncomfortably, if `a` nor `b` have any values at the moment, `if (a=b) then 1 else 2 fi` returns `1` while `if (a>b) then 1 else 2 fi` produces `Error, cannot evaluate boolean`.

The convention that `if` gives an error if `x` cannot be reduced to a Boolean value is only one possible convention among many for the "unknown" branch of an `if`. Macsyma and Mathematica make different provisions, with Macsyma allowing a choice of carrying the unevaluated `if` along, or signalling an error. See Appendix II on Conditional Expressions.

Maple's `evalc` forces evaluation over the complex numbers. It appears to provide several facilities intermixed. One facility attempts to split an expression into real and imaginary components in order to find a kind of canonical form for expressions. A second facility merely informs the system that additional numerical evaluation rules are available, such as `cos` of complex numbers. (M. Monagan explains `evalc` as complex expansion under the assumption that all symbols are real-valued.) More recent versions of Maple have taken some of `evalc`'s capabilities and added them to `evalf`.

At first sight the function `evaln(x)` seems quite strange — it is used to create a symbol that can be assigned a value. In the simplest case, it is the same as using single-quotes. You can use this to take some data and concatenate pieces together to "evaluate to a name". Although `evaln` has a few bizarre features, the notion of creating and installing a string in a system's symbol-table is handy.

The assignment operation in most languages implicitly uses "evaluate to a name" on the left-hand side of the assignment. Consider the sequence `i:=1, t[i]:=3, t[i]:=4`. The left-hand side of the expression `t[i]:=4` should be "evaluated" to the location for `t[1]`, not `3`, and not `t[i]`. Maple's penchant for the use of side-effects for assigning values to extra variables makes an explicit version of this operation handy. Thus, `divide(a,x+1,'q')` might test to see if `x+1`, divides exactly into the polynomial denoted by `a`. If so, `q` is assigned the quotient. In a loop, you might need a sequence of names for the quotients: `divide(a[i],b,evaln(t[i]))` where `i` is the index of a `for` loop.

The program `Eval`, quite confusingly from our perspective, is an inert operator used to represent an *unevaluated* polynomial and points to be used for evaluation. Apparently the (unstated) motivation is to make it faster to express results as residues in finite fields. `Eval(x^100-y^100,{x=3,y=4})` just sits there unevaluated, but computing that value `mod 11` returns `0`.

Maple's `evalr` implements a kind of interval arithmetic, here called "range arithmetic" to compute a confidence interval for a calculation. An associated function `shake` produces a interval to be fed into such functions.

The implementation details of `evalr` can be found, as is the case for much of Maple (everything but the kernel) by looking at the definitions which can be extracted in source code form from the running Maple system. In fact, the `evalr` system cannot work too well for the reasons given earlier: the Maple kernel assumes that two intervals with the same endpoints are identical, and that their difference is exactly zero.

In the versions of Mathematica prior to 2.2, the same error occurred; eventually the vagaries of the `Interval` structure were incorporated into the equivalent of the Mathematica kernel.

6.4 Mathematica

The underlying scheme for evaluation in Mathematica [12] is based on the notion that when the user types in an expression the system should keep on applying rules to it (and function evaluation means rule application in Mathematica), until it stops changing.

The evaluation strategy in Mathematica, as is typical with every computer algebra system, works well for easy cases. For more advanced problems, Mathematica's evaluation tactics, intertwined with pattern matching and its notion of Packages is more elaborate than most. It is clear that the evaluation strategy is incompletely described in the reference [12]; furthermore it appears it is never fully described in the Mathematica literature. Experimentation may be a guide.

It appears that the usual block structure expected of an Algol-like language is only partly simulated in Mathematica. The usual notion of contexts for bindings, as one might see in Pascal or C is actually simulated by another mechanism of Packages. Defining, setting or evaluating a simple symbol, say `x`, at the command level actually defines it in the Global (top level) Package. Its evaluation returns its binding `Global‘x`. Evaluation of a symbol defined but uninitialized in a `Module`, for example by `Module[{x}, ..x..]`, is actually the same as a symbol `Global‘x$12`. That is, a lexical context is implemented by mashing together names with sequentially generated numbers that are incremented at each use. There is also a `Block` construction, a remnant from an earlier attempt to implement block structure in Mathematica. The evaluation mechanism of "repeated evaluation until no change" pretty much defeated the local-name mechanism of `Block`: If the global value of `x` is 4, then `Block[{x},x]` evaluates to 4 (presumably in two stages: `x` evaluates to `x` in the outer block, and then `x` evaluates to 4).

Names can be defined in different `Packages`, perhaps nested, in which case inter-package visibility and remote-naming requires additional syntax of the form elaborate compound form `package1‘subpackage‘name`.

Evaluating a function or operator is quite elaborate; First, the name of a function is evaluated "until no change" to some symbol s. If s has a function definition, (or a re-writing rule, actually) with an appropriate number of arguments, those arguments are evaluated in turn, unless s has one of the attributes `HoldFirst, HoldRest`, or`HoldAll`. These indicate that some or all of the arguments are not to be evaluated immediately. They are set by using `SetAttribute`, as in the example below. Yet if a `Held` argument is evaluated once, it is evaluated "until no change". Thus confusingly, given

```
SetAttribute[foo,HoldAll];
z=4;
foo[x_]:=x;
bar[x_]:=x;
```

the two functions defined are indistinguishable: `foo[z]` and `bar[z]` will return 4.

But

```
foo[x_]:=x++
bar[x_]:=x++
```

are different. `foo[z]` returns 4 and sets `z` to 5. `bar[z]` is an error: one cannot increment a number.

It is also possible to prevent an evaluation by the judicious use of `Hold[]` and `ReleaseHold[]` as well as `Unevaluated[]` and `Evaluate[]`. Distinguishing between the semantics of these pairs seems pointless, since they all appear to be inadequate attempts to mimic the mechanism "quote" in an environment in which the "until no change" rule holds. It may be that a study of macro-expansion in Common Lisp or some other language in which these issues have been carefully designed and tested for a period of years, would provide a model for some other components of the Mathematica semantics.

Returning to the task at hand, assume now that the symbol s is a function definition and we've found one (the first in some heuristic ordering) rule that can be applied to rewrite the expression. If that fails, we try the next rule, etc. If all rules fail, then the expression is returned as s with its arguments.

To determine if some rule can be applied, we look at the possible definition structure for a function `f`. Even in a somewhat simplified explanation, we must deal with at least the following kinds of cases (we give prototypical examples):

1. `f[x_,y_]:=` for the usual parameter binding of two arguments.
2. `f[[x_,y_]]:=` or any other f[g[...]] for parameter destructuring.
3. `f[x_foo]:=` for explicit self-descriptive manifest-type checking.
4. `f[x_?NumberQ]:=` for implicit type-checking by predicate satisfaction.
5. `f[x_,1]:=` for special case arguments.
6. `f[a,1]:=` memo function for a particular set of arguments.
7. `f[x__]:=` flexible patterns for one or more arguments.
8. `f[x___]:=` for zero, one, or more arguments.
9. `f[x_]:=...` , `f[x_]:=..` it is possible to have multiple (even conflicting) definitions.
10. `g/:f[...g[...]...]:=` which define "uprules" that alter the meaning of `f`, but only if one of the arguments of `f` has a `Head` that is `g`.

A brief explanation of the uprule is probably warranted: This is a rule for rewriting `f`, but keyed to the evaluator noticing that `g` is in the top level of arguments to `f`. This is an optimization to prevent slowing down common operators where `f` is say `+` or `*`. Cluttering these common operators

with rules (say, to deal with the sum and product of a user-defined introduced function) would lead to inefficiencies.

Using the fixed-point philosophy throughout the system (not just at command level as in Maple) requires Mathematica to efficiently determine that when a rule is attempted, that in fact no change has happened (because such a change could trigger further rule application). Just as Maple falls short in its use of "option remember", Mathematica also appears to hold on to outmoded values. Mathematica applies some clever and apparently non-deterministic heuristics to determine this no-change termination condition. Because it is possible to change the global state of the system by rules that *fail* as well as by rules that succeed, the heuristic can easily be subverted. While we show, by deliberate means below, how to do so, the casual user can avoid it by using only simple rules where no side-effects on global state are possible if the rule fails. (This may not be entirely obvious, of course).

Here is a definition of a Mathematica function `g`:

```
i=0;
g[x_]:= x+i /; i++ >x
```

The two allegedly equivalent expressions `{g[0],g[0]}` and `Table[g[0],{2}]` result in `{g[0], 2}` and `{g[0], g[0]}` respectively.

Furthermore, Mathematica can be easily fooled into thinking the system has changed some dependent structure and thus will spend time re-evaluating things without effect. For example, after setting an element of an array `r`, by `r[[1]]=r[[2]]` the system must check that no rules are newly-applicable to `r`. This depends on how many elements there are in `r`. If `r` has length 10 this takes 0.6 ms., but at length 100,000, some 433 ms.[6]

There are additional evaluation rules for numerical computation in which `Accuracy` and `Precision` are carried along with each number. These are intended to automatically keep track of numerical errors in computation, although their failure to do so is one problem noted by Fateman [6].

Some expressions that are supposed to be purely floating-point (real) are "compiled" for rapid evaluation. This is useful for plotting, numerical quadrature, computing sound waves, and solving differential equations. The evaluation of compiled functions provides another set of semantics different from the usual arithmetic. This seems to be in a state of flux as versions change. In at least one version, the temporary excursion of a real-valued function to use a complex-valued intermediate result causes problems.

Evaluation of expressions involving certain other kinds of expressions, among them real intervals, and series, also seem to have special treatment in the Mathematica kernel. This must be handled rather gingerly. Consider that `O[x]^6-O[x]^6`, a series expression, is not zero but is "equal" to `O[x]^6` and `Interval[{-1,1}]-Interval[{-1,1}]` is not zero either, but `Interval[{-2, 2}]`.

6.5 REDUCE

The REDUCE system [11] uses a model of evaluation similar to that in Lisp, a language in which it has historically been implemented, although a C-based version now exists. REDUCE has two modes, The first is called *symbolic*, and consists of a syntactic variant of Lisp with access to the REDUCE library of procedures and data structures. This provides an implementation language level for the system-builder and the advanced user. The second mode is called *algebraic*, in which the user is expected to interact with the system. Among other features, unbound variables can be used as symbols, and undefined operators can be introduced. In both modes there is a general infix orientation of the language, but the programming and expression semantics are still generally based upon the Lisp model of recursively traversing a tree representing a program, evaluating arguments and applying functions, but with resubstitution until the expression being handled ceases to change. The simplification process is a reduction to a nearly canonical form, and subject to a certain number of flags (`exp, gcd`). A major model for the use of REDUCE is for the user to supply a number of rules that are defined via (`let` and `match`) statements, and then interact with user input. This has both the advantages and disadvantages stated earlier concerning rules.

The REDUCE system is admirably brief, at least if one ignores the size of the underlying Lisp system, and avoids some of the distressing aspects of more elaborate systems. The trade-off is that the REDUCE notation is somewhat more distant from mathematical notation, and some of the advanced capabilities of the system are available only after loading in modules from the substantial library.

6.6 Other systems

There are a number of new systems under development. Space does not permit comparison here, but we expect that to the non-expert, each appears to have an evaluation strategy similar to one or more described above.

7 Boundaries for Change

Various systems take different approaches in allowing the user to alter the course of evaluation.

Within the bounds of what can be programmed by the user, Maple provides some handle on the evaluation task: the code for evaluation is in part accessible, and distributed as properties of the operators. A similar argument can be made for *user-extended* parts of Mathematica. That is, one can specify rules for new *user-introduced* operators. In Maple or Mathematica one has rather little chance to intervene in the proprietary kernel of the system. Since so much more of the system in Mathematica is in the kernel, it makes changes of a fundamental nature rather difficult.

Macsyma's user-level system has similar properties to that in Mathematica, both with respect to adding and specifying new operators and changing existing ones. However for nearly any version of Macsyma (and REDUCE), it is possible by means of re-defining programs using Lisp, to change the system behavior. Although this is rarely recommended, a well-versed programmer, aided by available source code, has this route available. Such alternation is error-prone and risky since a programmer may inadvertently violate some assumptions in the system and cause previously working features to fail.

An example mentioned previously that causes problems in any of these systems, the correct implementation of an `Interval` data type, effectively cannot be done without kernel changes, since intervals violate the rule that $x - x = 0$. (According to interval rules, $[a,b]-[a,b] = [a,b]+[-b,-a] = [a-b, b-a]$.)

Axiom would simultaneously have less formal difficulty, and perhaps more practical difficulty handling intervals. I

[6] times for version 2.2 on a Sparc-1+ workstation.

suspect that such an algebraic system that violates $x - x = 0$ cannot inherit any useful properties of the algebraic hierarchy. Thus a new set of operators would have to be defined for intervals, from `+` to `cos` to `integrate`. This has the advantage of a relatively clean approach, but on the practical side, it means that many commands in the system that previously have been defined over (say) reals, and might be useful for intervals will require explicit reprogramming. The general rule that $f(X)$ for X an interval is $[\min_{x \in X} f(x), \max_{x \in X} f(x)]$ cannot be used because it is not sufficiently constructive.

Plausible goals for any scheme that would modify an evaluator are

1. It must leave intact the semantics and efficiency of unrelated operators (including compilation of programs involving them).
2. It must reserve natural notations.
3. It must display an economy of description.
4. It must, to the greatest extent possible, allow efficient compilation of programs using the modified evaluation.

8 Summary and Conclusions

From the view of studying programming languages, there are many well-understood "evaluation" schemes based on a formal model and/or an operational compiler or interpreter and run-time system. Traditional languages in which the distinction between data and program are immutable can be described more simply than the languages of computer algebra systems.

Among "symbolic" language systems where the data–program dichotomy is less clear, Common Lisp is rather carefully defined; the semantics of computer algebra systems tends to be described informally, and the semantics generally change from time to time.

Compromises in mathematical or notational consistency are sometimes submerged in considerations of efficiency in representation or manipulation.

Is there a way through the morass? A proposal (eloquently championed some time ago by David R. Barton at MIT and more recently at Berkeley) goes something like this: Write in Lisp or another suitable language[7] and be done with it! This solves the second criterion of our introductory section. As for the first criterion of naturalness – let the mathematician/user learn the language, and make it explicit. If the notation is inadequately natural, perhaps a package of "notational context" can be implemented for that application area on top of the unambiguous notation and semantics.

Providing a context for "all mathematics" without making that unambiguous underpinning explicit is a recipe that ultimately leads to dissatisfaction for sophisticated users.

What makes a language suitable? We insist that it be carefully defined. Common Lisp satisfies this criterion; the (much simpler) Scheme dialect of Lisp might do as well[8]; even a computer algebra systems language could work if it were presented in terms of unambiguous, aesthetically appealing, and consistent specifications.

Among the more appealing aspects of Lisp and related languages is that a clear distinction between `x` and `(quote x)` which is also denoted by `'x`. Evaluation is done by argument evaluation (one level), or by macro-substitution of parameters, or by explicit calls to `eval`, `apply` or `funcall`. The scope of variables, etc. are carefully specified by Lisp.

Another appealing although complicating aspect of Common Lisp is the elaboration of name-spaces (via its `package` concept). The relationships possible by importing, exporting, and shadowing names in a large collection of programs from potentially different sources is a welcome relief from systems in which arbitrary naming conventions must be imposed on programmers just to keep the cross-talk down to a low level. Mathematica's package notion may have been inspired by this development.

A minor variation to Lisp's evaluation – to avoid reporting certain error when a symbol is used unquoted, is used in MuLisp, a dialect of Lisp that supports the CAS Derive [9].

Consider a version of Lisp that has a modified `eval` that is exactly like `eval` in almost all respects except that errors caused by unbound variables or undefined functions result in "quoted" structure. Such a version of Lisp can be written as an interpreter in Lisp, or built within Common Lisp by altering the evaluator. Such an alteration makes it difficult to find true programming errors, since there is a tendency for erroneous input or programming errors to result in the construction of huge expressions. This crude model of a computer algebra system, among other consequences, allows any result or any argument to be of the type "unknown symbolic". It may be that a formalization and extension of this interpreter can serve as a guide for variations on evaluation.

An alternative view as to how one should construct of large systems that has been promoted recently is that of object-oriented programming. Indeed, writing certain computer algebra programs in Common Lisp's object system (CLOS) is somewhat more convenient than otherwise. The hierarchy of classes, coercions and definitions of methods that are needed for writing computer algebra can to a large extent be mirrored by CLOS. Work by R. Zippel [13] takes this view. The demands of computer algebra seem, however, to strain the capabilities of less sophisticated systems. In fact, Newspeak's multiple-generic functions [7] (where the types of all the arguments, not just the first) determine the method to be used, were adopted by CLOS, and are particularly handy[9].

Variations on the symbolic-interpreter model for CAS evaluation have dominated evaluation in the past; it seems that an object-oriented view may dominate thoughts about systems for a bit more time; perhaps a tasteful combination of the two will emerge in the future.

We have come to believe that the role of a computer algebra system is to make available those underlying algorithms from concrete applied mathematics, clearly specified, that might be useful to the experienced and demanding user of symbolic scientific computing. **Such an explicit recognition of the needs of the** *application programmer* **as well as the system builder is key to providing facilities that will solve important problems.** An application programmer (perhaps with the help of a system-

[7] Newspeak [7], Andante, were experimental languages developed at the University of California at Berkeley for writing computer algebra systems based on an algebraic mathematical abstraction that embodied most of what people have been trying to do. AXIOM's base language is similar in many respects.

[8] The usual criticism of Scheme is that it sacrifices too much efficiency for purity of concept.

[9] Simpler object-oriented systems where, in effect, the type of only one argument is used for determining the meaning of an operation, seem to defer but not eliminate painful programming.

building expert) has a chance of providing —in a particular domain—a natural, intuitive notation. These specialized "mini-languages" may be clustered in libraries, or may be stand-alone programs.

Perhaps if there is a lesson to be learned from the activity of the last few decades, it is this: For computer scientists to provide at one fell swoop a natural notation and evaluation scheme for *all* mathematicians and mathematics is both overly ambitious and unnecessary.

9 Acknowledgments

Thanks to R. D. Jenks, Keith O. Geddes, and M. Monagan, and Jeffrey Golden for comments on evaluation in AXIOM, Maple, and Macsyma. This work was supported in part by NSF Infrastructure Grant number CDA-8722788 and by NSF Grant number CCR-9214963.

References

[1] Abdali, S. K., Cherry, G. W., and Soiffer, N. Spreadsheet computations in computer algebra. *ACM SIGSAM Bulletin 26*, 2 (Apr. 1992), 10–18.

[2] Brownston, L., Farrell, R., Kant, E., and Martin, N. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Addison-Wesley, 1985.

[3] Buchberger, B., Collins, G., Loos, R., and Albrecht, R., Eds. *Computer Algebra: Symbolic and Algebraic Computation.* Springer Verlag, 1983.

[4] Davis, M. *Computability and Unsolvability.* McGraw-Hill, 1958.

[5] Fateman, R. J. Macsyma's general simplifier: Philosophy and operation. *Proc. 1979 Macsyma Users Conference, Washington, D.C.* (1979), 336—343.

[6] Fateman, R. J. A review of Mathematica. *J. Symbolic Comp. 13*, 5 (May 1992), 545—579.

[7] Foderaro, J. K. *The Design of a Language for Algebraic Computation Systems.* PhD thesis, Univ. of Calif. at Berkeley, 1983.

[8] Gradshteyn, I. S., and M.Ryzhik, I. *Table of Integrals, Series, and Products*, 4th ed. Academic Press, 1980.

[9] Inc., S. W. *DERIVE User Manual version 2.* Soft Warehouse, Inc, Honolulu, Hawaii, 1992.

[10] Jenks, R. D., and Sutor, R. S. *AXIOM: the Scientific Computation System.* NAG and Springer Verlag, NY, 1992.

[11] MacCallum, M. A. H., and Wright, F. *Algebraic Computing with REDUCE.* Oxford University Press, 1991.

[12] Wolfram, S. *Mathematica: A System for Doing Mathematics by Computer*, 2nd ed. Addison Wesley, 1991.

[13] Zippel, R. The Weyl computer algebra substrate. Tech. Rep. 90-1077, Dep't of Computer Science Cornell Univ., 1990.

Appendix I: Rule Ordering

There are many options to rule ordering, and a transformation may be successful with one order but lead to "infinite recursion" with another.

The exact nature of pattern matching and replacement need not be specified in the discussion below. Conventionally, patterns and their replacements would have zero or more "pattern variables" in them, and there might be associated predicates on these variables.

Given rules $r_i := p_i \rightarrow e_i$, $i = 1, \cdots, n$ where p_i and e_i are in general tree descriptions, apply the (ordered) set of rules $\{r_i\}$ to a tree E.

Scheme 1: Starting with $i = 1$ apply rule r_i exhaustively by trying it at each node of the tree E, explored in some fixed order (let us assume prefix, although other choices are possible). If the rule r_i applies, (namely, an occurrence of p_i is discovered), then replace it with e_i. Continue to the next subtree in the transformed E. When the initial tree is fully explored, then proceed to the next rule ($i := i + 1$) and repeat until all rules are done.

Scheme 1a: Halt at this time.

Scheme 1b: Start again with $i = 1$ with the transformed tree E and repeat again until a complete traversal by all rules makes no changes.

Scheme 1c: In case the tree keeps changing, repeat only until some maximum number of iterations is exceeded.

Variants to scheme 1a. When a rule r_j succeeds at a given position, immediately attempt to apply rules r_k for $k > j$ or $k \geq j$ to its replacement.

Scheme 2: Starting with the root of the tree E (or using some other fixed ordering), and starting with the first rule ($i = 1$) try each rule r_i at the initial node. If they all fail, continue to explore the tree in order. If some rule r_j applies, then replace that node p_j by e_j. Then continue to the next subtree in the transformed E until it is fully explored.

Scheme 2a: Halt at this time.

Scheme 2b: Starting with the root of the tree E repeat until there are no changes.

Scheme 2c: Repeat until some maximum number of iterations is exceeded. Variants to scheme 2a: When a rule r_j succeeds at a given position, immediately attempt to apply rules r_k for $k > j$ or $k \geq j$ to its replacement.

Heuristics: Some rules "shadow" others. Re-order the rules to favor the specific over the general. Use partial matching (or failure) of one pattern to deduce partial matching (or failure) of a similar pattern (e.g. commutative pattern matching can have repetitive sub-matches.)

In any of these schemes there is typically an implicit assumption that the testing of the rules' patterns is deterministic and free of tests on global variables, and thus once a pattern fails to match it will not later succeed on the same subexpression. In some systems the replacement "expressions" are arbitrary programs that could even redefining the ruleset).

Several application schemes were implemented in Macsyma, using different sequencing in the expression and through the rules. If only one rule is used (a common situation) several of the variations are equivalent. Mathematica has two basic variants of scheme 1, `ReplaceAll` and `ReplaceRepeatedly`, which in combination with mapping functions and a basic `Replace` provide additional facilities. In fact, elaborate rule schemes are rarely used for several reasons. The pattern-specification language and the manner of matching is already difficult to understand and control, and somewhat

separated from the major thrust of the language. Rules that do not converge under essentially *any* sequence are particularly difficult to understand. Especially for the naive user, it is more appealing to attach rules in Macsyma to the simplifier [5], or in the equivalent Mathematica form, to particular operators, than to use them in a free-standing rule-set mode.

Appendix II: Conditional Expressions

Consider the construction "if $f(x)$ then $a(x)$ else $b(x)$." As a traditional programming language construct it is clear that $f(x)$ should evaluate to a Boolean value *true* or *false*, and then the evaluation of either $a(x)$ or $b(x)$ must provide the result. It is quite important that only (and exactly) one of them is evaluated, for the purposes of reasoning about programs. If the evaluation of $f(x)$ provokes some error then the locus of control is directed elsewhere.

Let us assume now that these cases do not hold. We must come up with a possible CAS alternative for the case: $f(x)$ evaluates to g, a variable or (in general) an expression which is not known to be true or false.

1. We could insist in this case that anything non-false is true, and evaluate the $a(x)$ branch.

2. We could insist that this is an error and signal it as such.

3. We could defer the testing until such time as it could be determined to be true or false (the example below is somewhat hacked together to simplify the concept of scope here):

```
x:=3;
r:= if (x>y) then g(x) else h(y);
```

could result in

```
deferred_if  (3>y) then
  eval(substitute(x=3, g(x))) else eval(h(y))
```

If r is later re-evaluated, say with

```
y:=2;
eval(r) --> g(3)  evaluated
y:=4;
eval(r) --> h(4)  evaluated.
```

4. We could defer the testing but be less careful with the scope of variables, as is apparently done by Maple (Vr2)'s `if` construct: the scope of all variables in the deferred evaluation is taken to be that of the dynamic scope, so the meaning of the x in the Boolean expression $(x > y)$ could be different from the x in the $g(x)$`then` clause.

Macsyma allows the specification of several different options here, depending upon the setting of `prederror`. It also has a a program that may stop and ask the user for an opinion on logical expressions if it can't deduce the value. Mathematica has a version of the `If` with an extra branch, for "can't tell" although perhaps it should have yet another for "evaluation of the boolean expression caused an error".

Appendix III: A More Symbolic Lisp Evaluator

It is possible to alter the semantics of evaluation for Common Lisp by the use of two "hooks" (`*evalhook*` and `*applyhook*`) so that expressions which are executed as programs through the evaluation mechanism are subjected to alternative interpretations. It is, for example, possible to substitute a version of this more-symbolic evaluator for the normal system evaluator. This does not affect the speed or efficiency of compiled code except that errors in such code will not be given symbolic treatment. Thus it would be advisable to check, before entering *compiled* code that (say) assumed numeric inputs that the inputs were in fact appropriate. In some circumstances explicit checks on arguments are unnecessary because the compiler can determine the correctness of inputs.

For presentation here we display instead a mini-evaluator, not based on the use of the evaluator hooks. Consistent with a long tradition in Lisp, we provide a version of this Lisp by displaying an interpreter in Lisp.

The intent of the interpreter is to soften the requirements of Lisp in two fashions

1. Instead of giving an error for an attempt to refer to the value of a symbol when that symbol is unbound, the evaluator returns the symbol itself. It is as though `'x` instead of `x` had appeared.

2. If `(f x y ...)` is to be evaluated, but it provokes an error, the value returned is instead `(f a b ...)`. where `a` is the result of evaluating `x` etc. It is as though `(cons 'f (list x y ..))` had appeared instead of `(f x y ...)`.

For example: with `x` bound to 3, `y` unbound,

```
(mineval '(* (+ x (+ x 1)) (+ x y z)(f x)(f y))) -->
        (* 7 (+ 3 Y Z) (F 3) (F Y))
```

This is not completely effective; it assumes that the functions, at least those that provoke errors, have no side effects. As noted in the text, one effect is that the innocent "user" of such a system need not be quite as attentive as previously. Computing `(diff exp x)` when `(diff exp 'x)` when should have been typed, will not be a problem. Debugging incorrect programs becomes more difficult because such programs will often not halt as one might expect.

Normal correct lisp will be correctly evaluated as usual.

```
(defun mineval(form)
  ;; This is a mini-evaluator for common lisp that
  ;; heads-off some of the error handling and
  ;; results in a more "symbolic" evaluation.
  ;; Many complexities of CL are merely shunted
  ;; off to eval.

  (cond
   ;;form is a self-evaluating something,
   ;;like a number.
   ((constantp form)  form)

   ;; form is s a symbol; something to be evaluated
   ;; if it has a value.
   ((symbolp form) (if (not (boundp form)) form
                       (symbol-value form)))

   ;; form's an apparent function application
```

```
((consp form)
 (let ((head-fun-type (fboundp (car form))))
   (case (type-of head-fun-type)
     ;;type-of is somewhat implementation
     ;; dependent but this clause should
     ;; intercept IF, COND, etc.
     ;; Different systems may denote the
     ;;  type-of IF differently.
     ((excl::special-function macro-function)
      (handler-case (eval form)(error () form)))

     ;;a "normal" function that evals its args

     ((function compiled-function)
      (let ((args (mapcar  #'mineval (cdr form))))
        (handler-case
         (apply (car form) args)
         (error () (cons (car form) args))
         )))

     ;; At this point form is something else.
     ;; If the operator is not a defined function
     ;; just return form with args evaled.
     ;; Any other cases we evaluate and return
     ;; unevaled if an error occurs.
     ;; (and hope that no side effects happened)

     (t
      (let ((args (mapcar #'mineval (cdr form))))
        (if (null head-fun-type)
            (cons (car form) args)
          (handler-case (eval form)
                        (error () form)))))))))
```