ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

David N. Turner (Ed)

Functional Programming, Glasgow 1995

Proceedings of the 1995 Glasgow Workshop on Functional
Programming,
Ullapool, Scotland, 10-12 July 1995

Paper:

Algebraic Formula Manipulation in a Functional Language: A First Attempt

Antony J.T. Davie

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

Algebraic Formula Manipulation in a Functional Language: A First Attempt

Antony J.T. Davie¹

School of Mathematical and Computational Science, St. Andrews University
St. Andrews, Fife

Abstract

This paper represents the author's attempt to come to grips with some of the practical problems that arise out of conventional algebraic manipulation and leads to a discussion of the possibilities of generalising algebraic concepts such as commutativity and associativity to other algebraic systems.

An account is given of a class instance providing manipulation and simplification of 'school' algebraic formulae. A concept of *variability* is introduced which controls the order in which terms and factors appear in sums and products. Rules based on the commutative, associative and distributive laws of algebra, for simplification of sums, products and a few basic operators are presented.

Some discussion is given of how an advanced class system could possibly be of help in general algebraic simplification problems.

1 Introduction

A number of algebraic manipulation systems have become popular in recent years. Perhaps the most well known are Maple[1,3,9] and Mathematica[13]. At the end of the paper relevant properties of a lesser known but important system, Axiom[7], is discussed. All these systems include fully-fledged programming languages which allow the manipulation of formulae as a basic (but not primitive) type. These languages also provide pattern matching though this is through an explicit `match` function in Maple and Axiom uses it for specifying rewrite rules rather than separating cases in function definitions. Mathematica in particular has a quite functional flavour though it makes essential use of imperative features and functions are first order and strict². It should be possible to use the class concept to allow formulae to be manipulated in Haskell or Gofer in a similar fashion. The first task of this paper is to provide a type `Formula`, providing instances of the arithmetic operators for Gofer's class `Num`.

In fact this is perhaps not so straightforward as it at first seems because it is difficult, at least for a non-expert in this field, to come by details of the algorithms used to implement the various primitives of formula manipulation. The systems mentioned above are proprietary software and a great deal of commercial sensitivity prevents many of the algorithms from being published in the public domain. Two possible exceptions to this secrecy lie in references [2] and [6] which describe more advanced techniques than those described here. The first part of this work represents a possibly naive beginner's attempt to provide some of the simpler of the primitives needed for formula manipulation and is still under development. It is hoped that such a system will be useful in the public domain and the code is freely available.

In numerical analysis, the three processes of differentiation, interpolation and integration are respectively difficult, moderate and easy. Differentiation is difficult because small changes in the data can lead to large changes in the differential coefficient. At the other end of the scale, quadrature gives accurate results even if the data are not very accurate. In formula manipulation, the order of difficulty is the other way round. Differentiation is easy symbolically, but integration is a dark art as many will remember from school and university³. In between these two lie simplification which might be said to be the symbolic equivalent of interpolation. Perhaps it would be more accurate to say that substitution of values is equivalent to interpolation, but we usually have to simplify after substitution. In this first try, no attempt at symbolic integration is made. The easy options of differentiation and simplification are tackled and the latter is still under development. It was further decided, for simplicity's sake, to use the cut down class of numbers, `Num` which is provided by Gofer, rather than tackling the fully fledged number systems of Haskell.

¹email: `ad@dcs.st-and.ac.uk`

²Both languages have facilities for explicit delaying and forcing.

³ There is however an algorithm for symbolic integration which will either integrate a formula (consisting of 'well-know' functions) or tell the user that it can't be done[4].

The rest of the paper is organised as follows: Section 2 describes the basic type `Formula` used to represent algebraic expressions internally. Section 3 discusses the concept of variability, used for commuting terms and factors into a canonical order. In section 4, the rules governing the rewriting of formulae into simplified form are discussed. Section 5 briefly mentions differentiation and substitution. In section 6 we summarise our exploratory simplification system and finally, in section 7, we briefly discuss how an advanced class system might allow the extension and generalisation of these somewhat *ad hoc* techniques to other algebra systems.

I am grateful to many for helpful suggestions but especially for those made by Rita Loogen, Paul Hudak and Andy Gill.

2 The type `Formula`

Before we can indulge in simplification, we must ask what it means for a formula to be simplified. Is $(x-a)(x+a)$ simpler than (x^2-a^2) or vice versa? The view taken here is that polynomials should be multiplied out and like terms gathered together, so that the second of the above formulae is regarded as simpler than the first. This is hardly a definition, but it gives a flavour of what is meant. In fact it is very difficult to give a satisfactory definition except in this pragmatic way. One approach is to say that a formula is simplified if I say it's simplified or if my program produces it as the result. But this is similar to saying that a language's syntax and semantics is defined by the actions of a compiler. What we can say is that a simplified expression is produced by a reduction process and that simplified expressions are canonical forms of unsimplified ones. A much more general approach could be taken of course. Why should the algebra of formulae be restricted to 'classroom algebra'? This certainly seems to be the approach taken by Maple and Mathematica both of which contain large numbers of basic functions, some of them pretty *ad hoc*, for treating conventional formulae and it is what has been attempted here. In the last section, however, we briefly describe a more general approach.

This implementation of formulae centres round the premise that no formula should ever be allowed into the system which is not in simplified form. Thus we are dealing with a constrained type, not a free one and each formula when constructed may undergo a rewriting process until it is in a canonical form. Miranda's (obsolete) *laws*[11] allow the rewriting of constructed formulae at the point of creation and might seem to be what is wanted here. The related idea of *views*[12] might have been useful if it had not been abandoned from Haskell. It is clear that care must be taken to ensure that the rewriting of formulae into simpler forms terminates.

The free type of which our constrained type is a subset is defined as follows:

```
data Formula = Var String |
              Const Float |
              Sum [Formula] |
              Prod [Formula] |
              Power Formula Formula |
              Ln Formula |
              Exp Formula |
              Indet
```

`Var` constructs variables from their name and `Const` constructs constants. `Sum` and `Prod` construct sums and products respectively from their lists of terms and factors. `Power` raises one `Formula` to the power of another and `Ln` and `Exp` represent the Napierian common functions. More will be said later on about the common functions. Finally `Indet` represents indeterminate forms such as those obtained by dividing by zero or taking the logarithm of a negative number.

The lack of a subtraction constructor will probably be remarked on. Expressions with subtraction are represented internally by using a factor of -1 e.g. $y-x$ is represented by

```
Sum [Var "y", Prod [ Var "x", Const (-1.0) ] ]
```

Similarly division and reciprocation use an exponent of -1. These representations make it easier to gather terms and factors together.

3 Variability

A common way of constraining a free type is to provide for each of the constructors a rule which prevents any objects not in the type from occurring by rewriting such objects to ones which are in the type.

The main interest in `Formula` as a constrained type lies in the fact that addition and multiplication commute allowing us to reorder terms and factors. This leads to the restriction of lists of terms and factors in `Sum` and `Prod` to be *ordered by variability*. When humans write sums such as $x+a$, convention dictates that the variables tend to

come near the beginning of the sums and the constants near the end. On the other hand products tend to be written the other way round with constants before variables, e.g. $a * x$. Internally, we shall order the lists of both terms and factors with the 'more variable' formula first and the constants later. This ordering allows us, besides giving us canonical forms of Formulae, easily to gather *like terms* together. The idea of variability is similar to one found in [5].

In order to facilitate these ideas, an enumerated type

```
data Compare = Cnst | Vrbl | Comparable
```

is defined which represents the result when two formulae are compared for variability. If the first is 'more constant' than the second, the result of a comparison (using function `comex`) is `Cnst`, if more variable than the second, the result is `Vrbl`, and if the two formulae are candidates for combination by gathering like terms, then `Comparable`.

Variability of formulae is defined as follows:

- Pure variables are ordered by their names' lexical ordering. This agrees with convention in that names (at least single letter ones) near the beginning of the alphabet are generally assumed to represent constants while those near the end are used for variables. The system could be parametrised with a variable ordering function or such a function could be included as a global ordering, but I have taken the simpler decision.
- Numbers are even less variable than variables. Negative numbers are more variable than positive and numbers of greater absolute value are more variable than those of less absolute value. This allows terms of polynomials to be kept in a standard order.
- Formulae constructed with the same constructor are compared by looking at their arguments in turn from left to right and the first one determines.
- Each constructor is assigned a variability in the order

```
Ln > Exp > Power > Prod > Sum
```

and when two formulae with different constructors are compared, the one with the more variable constructor is compared recursively with the first operand of the other.

These rules (which are not complete — see the Appendix for the complete definition of `comex`) make intuitive sense. They imply for instance that sums are more variable than their terms, products than their factors, variables than any of their powers and higher degree polynomials than lesser degree polynomials. The strangest consequence is that numbers appear at the end of products. But this is only the internal form of the formulae and the overloaded instance of `showsPrec` for formulae can be (and is) made to reverse the order of factors in the `Text` form of products.

But apart from their intuitive significance, there is a good programming reason for organising variability in this way namely, when rewriting constructed formulae into simplified forms, only the examination of leading terms in lists of arguments is required. This is discussed in more detail in the description below of the two merging functions, `mergeSums` and `mergeProds`.

An alternative method of organising constrained types for sums and products might have been to define binary constructors for each of them and then constrain the formulae on the spines of the trees to be in order of variability. However this would have turned out to be pretty well isomorphic to the constrained list type — as we shall see sums and products tend to get flattened — and moreover we would have lost the ability to use all the list functions in the Gofer or Haskell standard prelude.

4 The Simplification Functions

The definitions of the instances required to make `Formula` a member of the (cut down Gofer) class `Num` are as follows.

```
instance Num Formula where
    x+y      = summer[x,y]
    x-y      = summer[x,prodder[y,mone]]
    x*y      = prodder[x,y]
    x/y      = prodder[x,y ** mone]
    negate x = prodder[x,mone]
```

The functions `summer` and `prodder` both take lists of arguments to be added or multiplied respectively and each of these must be of the constrained type. They must be simplified already. These two functions are also public and are available to users of the system if they need to add or multiply longer lists of formulae. A small amount of unfolding leads to the revised definitions:

```
instance Num Formula where
  x+y = formSum (mergeSums (terms x) (terms y))
  x*y = formProd(mergeProds(factors x)(factors y))
  ...
```

where the real work is shown being done by two list merging functions, `mergeSums` and `mergeProds`. The function `terms` converts a `Formula` into a list (already sorted by variability) and `formSum` converts a list back to a `Formula`, usually a `Sum`. Similarly for `factors`, `mergeProds` and `formProd` which operate with respect to products.

4.1 Addition

`mergeSums` works as follows:

- As already described, more variable terms migrate to the head of the lists. Thus the *Commutative Laws of Addition* are applied.
e.g.⁴ $x + 3 + y \Rightarrow y + x + 3$
- If any of the arguments in the list are themselves `Sums`, these are flattened and each item in their argument lists is treated as a separate argument to `mergeSums`. In other words the *Associative Laws of Addition* are applied.
e.g. $(x + z) + y \Rightarrow z + y + x$
- Like terms are gathered together by factoring out the more variable factors. In other words the *Distributive Laws of +, ** are applied.
e.g. $x * y + (y + x) \Rightarrow y * (x + 1) + x$
(though it would be printed as $(x + 1.0) * y + x$). Note that the less variable factors remain distributed so that $y * 2 + x * 2$ does not get changed to $(y + x) * 2$. The next action may be to add another term involving y to the sum.
- Numbers are added when their values are manifest.
e.g. $x * 4 + y * 3 + x * 2 - y$ first $\Rightarrow y * (3 - 1) + x * (4 + 2)$ and then $\Rightarrow y * 2 + x * 6$.
- Terms equal to zero are cancelled out. Zero is the *Unit of Addition*.
e.g. $x + y - x$ first $\Rightarrow y + x * (1 - 1)$, then $\Rightarrow y + 0$ and then $\Rightarrow y$.
- If two or more terms remain, the result is a `Sum`. If one remains that is the answer and if none remains, the answer is 0 (`Const 0.0`). `Sum` is not applied to singleton or null lists. e.g. y , the result of $x + y - x$ shown above, is not represented by `Sum [Var "y"]`, but just by `Var "y"` and $x - x$ does not transform to `Sum []`, but to (`Const 0.0`).
- If an attempt is made to add `Indet`, the result is `Indet`.
e.g. $x + 0 / 0 \Rightarrow \text{Indet}$.

4.2 Multiplication

As is to be expected, `mergeProds` has many similarities to `mergeSums`. The commutative and associative laws are implemented in the same way, using a list sorted by variability. The unit of multiplication is cancelled except where only one term remains in a product. Numbers are multiplied out whenever possible. Apart from that:

- Less variable factors are distributed *inwards* by multiplying out. Here the *Distributive Laws* are being used to unfold formulae.
e.g. $(y + x) * 2 \Rightarrow y * 2 + x * 2$
- Like exponents are added. The *Distributive Laws of *, *** are applied.
e.g. $y ** 2 * e ** 3 * y * e ** (-1) \Rightarrow y ** 3 * e ** 2$

4.3 Exponentiation and Logarithms

It will have been remarked that there are two exponentiation constructors, `Exp` and `Power`. This means that strictly speaking there is more than one canonical form of some formulae, e.g. e^x — Is this represented by `Exp x` or by

⁴We use conventional algebraic notation to show examples of the rewrites, rather than using the internal form which is more cumbersome e.g.

```
summer[Var "x", Const 3.0, Var "y"] =
  Sum [Var "y", Var "x", Const 3.0]
```

`Power e x` where `e` is `Const 2.7182818...`? The reason for the slight overkill here is a purely pragmatic one, namely that `Exp x` is easier to handle than a formula with an embedded constant with special properties that have to be recognised; and moreover, when it comes to calculus, `Exp x` has considerably simpler properties. The differential coefficient, for instance, of $e^{f(x)}$ is

$$f'(x) * e^{f(x)}$$

while that of $g(x)^{h(x)}$ is

$$g(x)^{h(x)} * [h'(x) * \ln g(x) + h(x) * g'(x) / g(x)]$$

The latter does indeed simplify to the former when $g(x) = e$ but only after several reductions and pattern matching $g(x)$ against `e`.

The penalty that has to be paid, however, is that simplification rules for both `Power` and `Exp` have to be implemented. These lie in the functions `(**)` and `exp'`. (Neither name is included as a Class method for `Num` in `Gofer`. `(**)` is not part of `Gofer`'s standard prelude but `exp` is).

The rules governing the simplification of `Power`s, where either the base or exponent is 0, 1 or indeterminate are in the main turgid and trivial:

x^y	0	1	Indet	y
0	Indet	0	Indet	Indet
1	1	1	Indet	1
Indet	Indet	Indet	Indet	Indet
x	1	x	Indet	investigate further

But one is of particular interest. What is the value of 0^f ? This is easy if `f` is a manifest constant (0 if positive and indeterminate otherwise); but if it is not manifest what the sign the value of `f` is, we don't know. At the moment, `Indet` is always returned in such circumstances. Suffice it to say that this problem is recognised by `Maple` and `Mathematica` both of which allow users dynamically to place constraints, such as conditions about signs, to be put on variables. This is an area fraught with difficulty however as it leads to a whole algebra of derivation of inequalities and few systems make attempts to solve this problem in any automatic way.

The further investigation mentioned in the last box of the above table is detailed as follows:

- If the base is a `Sum` and the exponent a positive integer, repeated multiplication is performed
e.g. $(y + x)^3 \Rightarrow y^3 + 3 * x * y^2 + 3 * x^2 * y + x^3$
- If the base is a `Prod` the exponent is applied to each factor and the product of the results taken
e.g. $(y * x)^3 \Rightarrow y^3 * x^3$
- Numerical simplification is carried out where possible
e.g. $(-3)^{-2} \Rightarrow 0.11111$
- Exponents in products are multiplied
e.g. $(x^3 * e^x)^2 \Rightarrow x^6 * e^{2*x}$

There is a collection of rules for `Exp` as well, some of which interact with those already given:

- $e^{\text{Indet}} \Rightarrow \text{Indet}$, $e^0 \Rightarrow 1$
- Exponents with are `Sums` are factored out to products of exponents
e.g. $e^{x+y} \Rightarrow e^x * e^y$
- If the first factor of the exponent is a logarithm, the formula is rewritten to one with a `Power` in it instead.
e.g. $e^{\ln x * 2} \Rightarrow x^2$

For logarithms, the rules are:

- $\ln \text{Indet} \Rightarrow \text{Indet}$, $\ln 0 \Rightarrow \text{Indet}$, $\ln 1 \Rightarrow 0$
- Logarithms of products are turned into sums of logarithms
e.g. $\ln (x*y) \Rightarrow \ln y + \ln x$
- Logarithms of exponentials simplify in the obvious way
e.g. $\ln (e^x * x^2) \Rightarrow \ln e^x + \ln x^2 \Rightarrow 2 * \ln x + x$

5 Differentiation and Substitution

In a similar but much more straightforward way, each constructor of `Formula` leads to a rule for differentiation with respect to a variable and one for substitution for a given variable. These rules are not difficult and will not be given here as they can be gleaned from any textbook on calculus. They are sometimes complex — for instance the one for differentiation of a power given in the previous section.

Substitution is also a straightforward process and involves the recursive invocation of the simplification rules.

6 Summary of the Exploratory System

The system as it stands at present offers the following public interface to allow users to manipulate formulae:

- instances of `(+)`, `(-)`, `(*)`, `(/)`, `negate` and `fromInteger`⁵ for `Formula`
- `summer` and `prod` which respectively add and multiply lists of `Formulas`.
- `(**)`, `exp` and `ln`
- `(==)` and `showsPrec` instances for equality testing and converting `Formulas` to textual form
- functions for differentiation and substitution

The seemingly *ad hoc* collection of simplification rules have to a certain extent been arrived at by experiment. In general they tend to convert powers into products and products into sums, but it cannot be denied that, except for the obvious ones, they were difficult to arrive at, and there is little doubt that there is room for improvement. Unfortunately pragmatism does not always sit well with theory. One theory that will be further investigated is the confluence properties of the simplification rules[8] which decide whether the order in which they are applied makes any significant difference. It would also be useful to prove that the set of simplification rules always leads to termination, that the simplification process cannot get into a loop. Methods for proving termination often rely on the definition of a *valuation* of terms such that a rewrite always produces a term with a smaller value. If the valuation can be chosen in such a way that the values are all positive integers, termination follows obviously; but choice of such valuations is often requires quite subtle expertise[10].

The *ad hoc* nature of the rules suggests that it may be difficult to see how to allow users to add rules for new constructors (such as `Sin` and `Cos`) in a safe way. Rules in which a new function occurs as the outermost operator are possibly moderately easy to add

e.g. $\sin(x + y) \Rightarrow \sin x * \cos y + \cos x * \sin y$

but rules which interact with already existing operators and where the new operator occurs inside an expression would require alteration of existing rules and their delicate balance might be disturbed.

e.g. $\ln(\sqrt{x}) \Rightarrow \ln x / 2$
or $(\sin x)^2 + (\cos x)^2 \Rightarrow 1$

It is unclear how to integrate such rewrite rules into the current system at present. For the present, therefore, the system will be further developed in its current *ad hoc* manner.

The system in general requires tidying and a fully fledged Haskell class instance of `Floating` or `Complex` formulae needs to be made in the form of a module. Extensions to the formula concept need to be made. For instance it should be possible to have formulae with free names for generic functions. In addition the concept of variability may need to be extended when combined with total and partial differentials. It should be possible to mark variables in some way as dependent and others as independent. e.g. If $x + y$ is differentiated with respect to x , do we get $1 + \frac{dy}{dx} (y \text{ dependent of } x)$ or $1 + 0 (y \text{ independent of } x)$?

7 More Generalised Systems

The rules developed in the exploratory implementation presented here may seem somewhat *ad hoc*. However a careful examination will show that they are based on the structure of the particular algebra which is in common use by most scientists and engineers, what has been sometimes called 'school' algebra. The main commutative, associative and distributive laws are mirrored in the techniques and functions used for simplification. Haskell and, to

⁵These are the 'standard' operators for the simplified class `Num` used by Gofer. For non-Haskell speakers, `FromInteger` is an overloaded function which changes literal integers into whatever subject type is being designated as belonging to `Num`, in this case `Formula`.

a certain extent, Gofer have several classes of numbers such as Integral, Fractional and Floating. Clearly a fully fledged usable system capable of simplifying complex expressions should be built round a type system with at least these classes; but this would still be somewhat *ad hoc* and some kind of abstraction based on the algebraic laws that systems obey is needed.

Other algebras have different sets of laws. Boolean algebra, for instance, has two symmetric distributive laws where *and* distributes over *or* and *vice-versa*. Algebras will share properties and algorithms which simplify, say, distribution in one algebra may also work in another. If an attempt is to be made to generalise the system, it should be based on a class system which reflects a hierarchy of algebras. The *Axiom* algebraic computation system[7] is one that the author is studying at the moment. *Axiom* allows classes and class instances (though with different terminology — *Category* and *Domain* roughly correspond to Class and Type) in many ways similar to those found in Haskell and Gofer. In addition, however, users can *assert* that particular rules are satisfied by particular hierarchies of algebras. For example not only will any domain of category Ring have the operations "+", "-", and "*", but "+" will be asserted to be commutative and associative and there will be an identity element and additive inverses.

The assertions can be quite complicated and even conditional. For example it is possible to specify a parametrised domain of algebras representing Complex numbers over some subdomain R by exporting the information that it is a Ring and also a Field provided R is a field. Thus the ring Complex(Float) is a field but the ring Complex(Integer) is not since Integer is not a Field since it has no division.

Unfortunately, in its present state of development, the assertions made by users are not fully checked by the *Axiom* system. It would be quite possible to make an incorrect assertion which the system might act on (by providing a particular simplification rule, for instance) in an incorrect way. Clearly the ability to do such checks automatically leads us into the realms of automatic theorem proving or, at the very least, proof checking. Research by the author and his colleagues is moving in this direction.

8 References

1. Abell, M., J. Braselton, *The Maple V(R) Handbook*, Academic Press Professional, 1994, ISBN 0-12-041542-9
2. Akritas, A.G., *Elements of Computer Algebra*, Wiley, 1989, ISBN 0-471-61163-8
3. Char, B.W., Geddes, K.O., Gonnet, G.H., Benton, L.L., Monagan, M.B. & Watt, S.M., *Maple V Language Reference Manual* and *Maple V Library Reference Manual*, Springer-Verlag, 1991
4. Davenport, J.H., *Intégration algorithmique des fonctions élémentairement transcendentes sur une courbe algébrique*, Annales de l'institut Fourier 34, 1984, pp 271-276
5. Fletcher, J.G., *GRAD Assistant — A Program for Symbolic Algebraic Manipulation and Differentiation*, Lawrence Radiation Lab., University of California Livermore, Research report UCRL-14624-T, 1965
6. Harper, D., Wooff, C. & Hodgkinson, D., *A Guide to Computer Algebra Systems*, 1991, John Wiley & Sons, Inc., ISBN 0-471-92910-7
7. Jenks, R.D. & Sutor, R.S. *Axiom, The Scientific Computation System*, Springer-Verlag, 1992
8. Knuth, D. & Bendix, P., *Simple Word Problems in Universal Algebras* in *Computational Problems in Abstract Algebra*, ed. J. Leech, Pergamon Press, Oxford, 1970, pp 263-197
9. Kofler, M., *Maple V Release 3*, Addison-Wesley, 1994, ISBN 3-89319-765-6
10. Plaisted, D., *Equational reasoning and term rewriting systems*, in *Handbook of Logic in Artificial Intelligence and Logic Programming*, Oxford University Press, pp 274-367, 1993
11. Thompson, S., *Laws in Miranda*, University of Kent at Canterbury Computing Lab. Report 35, 1985 — also Proc. 1986 ACM Conf. on LISP and Functional Programming, MIT, 1986, pp1-12
12. Wadler, P.L., *Views: A Way for Elegant Definitions and Efficient Representations to Coexist*, Oxford Univ. Programming Research Group, 1985 — also Aspenäs Workshop on Implementation of Functional Languages, Göteborg 1985
13. Wolfram, S., *Mathematica, A System for Doing Mathematics by Computer*, Addison-Wesley, 1991

Appendix

This is the complete code for comparing two expressions for variability. The third argument is a piece of state acting as a default

```
comex :: Formula -> Formula -> Compare -> Compare

comex x          y          i | x==y  = i
comex (Const x)  (Const y) _        = comno x y
comex (Const _)  _          _        = Cnst
comex (Var x)    (Const _) _        = Vrbl
comex (Var x)    (Var y)   _ | x<y   = Cnst
                           | True  = Vrbl

comex (x@(Var _)) y          _        = comex x (firstArg y) Cnst
-- Non-Haskell experts.
-- See footnote for the meaning of @6
comex _          (Const _) _        = Vrbl
comex x          (y@(Var _)) _      = comex (firstArg x) y Vrbl
comex x y i | precedence x > precedence y
              = comex x (firstArg y) Cnst
              | precedence x < precedence y
              = comex (firstArg x) y Vrbl
              | True
              = comlist (args x) (args y) i

comlist []      []      i = i
comlist []      _      _ = Cnst
comlist _       []      _ = Vrbl
comlist (x:xs) (y:ys) i =
    if j==Comparable then comlist xs ys i else j
    where j=comex x y Comparable

comno x y | x==y          = Comparable
comno x y | x<0.0 || y<0.0 =
    if x<y then Vrbl else Cnst
comno x y
    =
    if x<y then Cnst else Vrbl

args (Sum xs)    = xs
args (Prod xs)   = xs
args (Power x y) = [x,y]
args (Exp x)     = [x]
args (Ln x)      = [x]

firstArg (Sum (x:_)) = x
firstArg (Prod(x:_)) = x
firstArg (Power x _) = x
firstArg (Exp x)     = x
firstArg (Ln x)      = x

precedence (Sum _)      = 3
precedence (Prod _)     = 4
precedence (Power _ _) = 5
precedence (Exp _)      = 6
```

Code for the rest of the system is available on application to the author.

⁶The notation $v@p$ where v is a variable name and p is a pattern is called an *as-pattern*. It allow one to use v as a name for the whole value being matched by p .