Mathematics and Algorithms for Computer Algebra

July 9, 2003

Abstract

This course will be mainly mathematics, some computer science and a little computing. Little or no essential use will be made of actual computer languages, although I may occasionally use Pascal, C, Lisp or REDUCE for concrete examples. The aim of the course is to provide an entry into the current research literature, but not to present the most recent research results.

The first half of the course (taught by me) will deal with *basic* mathematics and algorithms for computer algebra, primarily at the level of arithmetic and elementary abstract algebra, including an introduction to GCDs and the solution of univariate polynomial equations. This leads into the second half of the course (taught by Dr Jim Skea) on the more advanced problems of polynomial factorization, indefinite integration, multivariate polynomial equations, etc.

The first week provides an introduction to the computing aspects of computer algebra, and contains almost no mathematics. It is intended to show how the later theory can be implemented for practical computation. The second week provides a rapid but superficial survey of the abstract algebra that is most important for computer algebra. The next five weeks will build on this abstract basis to do some more concrete mathematics in more details, referring back to the basis established in the first two weeks as necessary.

At the end of each set of notes will be exercises, one (or more) of which will be assessed.

1: Introduction to Computer Algebra

1 Data types and tasks of computer algebra (CA)

We primarily want to compute with *expressions* such as

$$\frac{23x^2 + 4y - \sin(2z/3)}{15xy(1+z)}.$$

But let us start more simply, with just

$$23.$$

What is this? Conventional notation is horribly ambiguous. It could be

- an integer,
- a rational,
- a real,
- a complex,
- an integer modulo some $m > 23$.

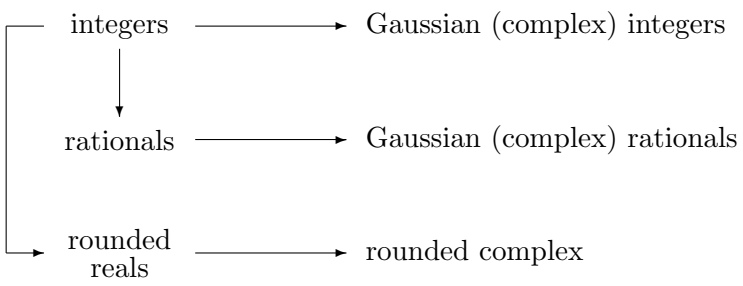
Does it matter? Yes, for two reasons:-

1. The operations defined on a set, or (loosely) the results of applying operations to elements of a set, depend on the set, e.g. over the integers 23/15 is not defined, whereas over the reals it is.
2. It affects the computational representation, because it is possible to represent any (finite) element of a discrete (i.e. countable) set *explicitly* and *exactly*, e.g. an integer or rational, but not a *general* element of a continuous (i.e. uncountable) set, e.g. an irrational real, which can be represented only *either* implicitly (i.e. symbolically) *or* approximately.

I will call an approximate representation of a continuous set a *rounded* representation – usually it will be a *floating-point* approximation of a real.

It is useful to identify the following *hierarchy of data types* for CA:-

0: Number domains (or coefficient domains)

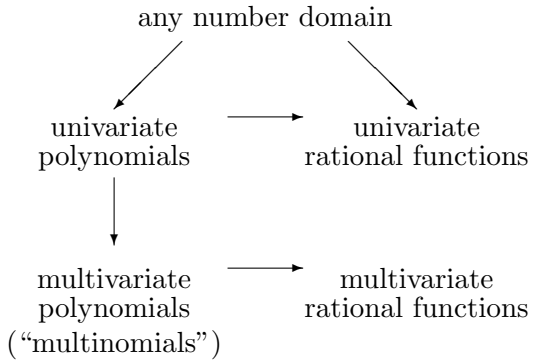


The geometry of the above and next figures shows the logical relationships among the domains, whereas the arrows indicate the likely relationship among their representations in practice.

The above can all be explicitly represented. Implicit symbolic representations of algebraic and transcendental irrational reals are also important, e.g. $\sqrt{2}$ and e. However, whilst these logically have the status of domain elements, they may in practice be represented as functions and variables (respectively).

At any point in a computation the number domain should be fixed and uniform (which may require type conversions).

1: Polynomials and rational functions Based on any specified number domain, the next level of structure is this:-



2: Arbitrary scalar expressions The next level of structure is to allow *functional forms*, i.e. function applications that are not evaluated but remain in the symbolic form

function(arguments).

The "function" is not allowed to be arithmetic, because that corresponds to the "rational function" level (1) already discussed above. The functional form has the status of a polynomial variable (or unknown). [REDUCE calls variables and functional forms all *kernels* – respectively simple and extended.] We have now reached the level of structure of our initial example.

3: Structured data types The final level of the data hierarchy consists of structured objects constructed from the above scalar data types, such as sets, lists, equations, arrays, matrices, vectors, tensors. A *list* is an ordered set with sequential access only (meaning that to access any particular element it is necessary to step through all the elements from the first until the required element is found). An *array* is an ordered set with indexing that allows random access (meaning that any element can be accessed directly and independently of the other elements via its index).

Not all CA systems explicitly support all these structures, and they may be represented internally in terms of each other; for example, in REDUCE only lists and arrays are fundamental, and both sets and matrices (!) are represented in terms of lists.

Note that sets, lists and arrays have no intrinsic properties other than their structure, whereas matrices, vectors and tensors have various additional arithmetic and algebraic properties.

The tasks of CA now follow naturally from the above data types, and are to perform all operations defined on them, namely:

- arithmetic operations on number domains, polynomials, rational functions and structured data types;
- symbolic operations of substitution (or equivalently composition), regrouping or reformatting, and accessing components;
- algebraic operations such as factorization, gcd computation, equation solving;

- pseudo-analytic operations such as differentiation and integration;
- series summation, series expansion and manipulation;
- higher-level applications, such as geometry.

In this course we will not consider symbolic operations, which are more computer science than mathematics, nor (much) the higher-level applications that require specialist knowledge of particular application areas.

2 The main CA systems: Maple, REDUCE, Derive, etc.

The main current general-purpose CA systems are Maple, REDUCE, Derive, Mathematica, Axiom and Macsyma. The oldest are Macsyma and REDUCE, which both first appeared around 1970, and the newest are Derive and Mathematica, which appeared around 1990. Derive runs *only* on MS-DOS machines; Maple and REDUCE run on many machines from fairly modest modern MS-DOS machines upwards; the other systems require more powerful machines. All but Derive provide a programming language that can be used to extend the facilities of the basic system, which has led to useful libraries of both developer- and user-supplied code.

All the systems support most of the scalar data types described above and (possibly via library routines) also most of the structured types. They also support most of the defined operations. They differ in the machines they run on, their user interfaces, their support of graphical output, their numerical capabilities, and their support for application areas (such as higher transcendental or "special" functions).

REDUCE, Derive, Axiom and Macsyma are written essentially in different versions of Lisp; Maple and Mathematica are written essentially in C. However, large parts of all the systems (except Derive) are written in their own programming languages, but they all have some kernel written in Lisp or C. [The most recent implementation of REDUCE uses Codemist Standard Lisp (CSL), which is itself written in ANSI C, so REDUCE can be considered to be implemented in C using Lisp internally as an intermediate language.]

2.1 What computational facilities are required to support computer algebra?

Clearly an underlying numerical capability to support the number domains, plus a symbolic capability to support the textual nature of general algebraic expressions, and flexible support for structured data. But probably the most important capability is *dynamic memory management*.

When computing with just numbers it is usually possible to allow enough memory for the largest likely number, and to reserve this memory for each of the numbers that needs to be retained at the same time. But one algebraic expression is generally *much* more complicated than just a number, and so would take *much* more memory. Moreover, it is difficult to predict the most complicated expression that can arise during the course of an algebraic computation. As an example of how an apparently simple expression can become much more complicated, consider

$$\frac{x^{10}-1}{x-1} = x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1,$$

in which the result of performing the division has many more terms than either the dividend or the divisor.

The only practical solution is to manage memory dynamically, i.e. from a pool of available memory, allocate memory to store data as necessary during the computation and return it to the pool when it is no longer needed. Dynamically allocated memory must be accessed via pointers (i.e. dynamic addresses).

C is a very good language for all these tasks, but it is fairly low level. Lisp already provides all the essential facilities automatically. Hence it makes sense to write an algebraic language processor (kernel) in C and implement a CA system in that kernel language, or to use Lisp as the kernel language, where the Lisp is implemented in C or some similar system language. The details make little real practical difference.

[Note that CSL-REDUCE is the *only* CA system that provides full source code, including the C code for the Lisp kernel.]

3 Data representations and their implementation

This section is based on REDUCE, but the essential ideas must apply to all systems. However, whereas REDUCE uses Lisp lists as the basis for nearly all its data structures, Maple uses hash tables, for example.

3.1 Integers

Computer hardware provides storage of, and arithmetic on, integers up to some fixed size, typically $2^{31} - 1$. This size limit assumes a *word* size of 32 bits, where by "word" I mean here the largest amount of data that can be processed in one step. Computation with integers having this size limit is *fast*.

However, this limit is (much) too small for CA, because simple computations with expressions involving small numbers can easily produce very large numbers. Consider expanding $(x+1)^{100}$. The largest coefficient in the binomial expansion is in the middle, and is ${}^{100}C_{50}$ or $\binom{100}{50}$. This is

$$\frac{100!}{50!50!} \approx \frac{100^{100}}{50^{50}50^{50}}$$

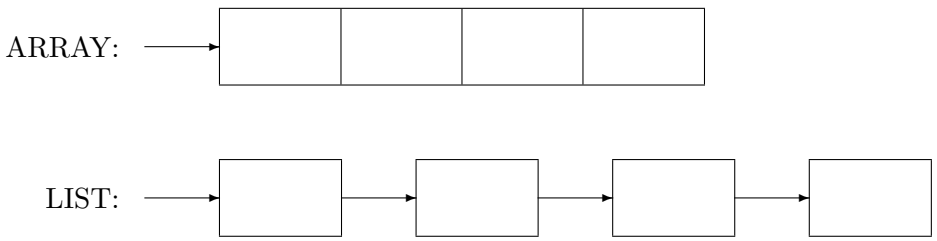
(by a crude application of Stirling's asymptotic formula for the factorial), which evaluates to

$$\frac{(2 \times 50)^{100}}{50^{100}} = 2^{100} \approx 10^{30},$$

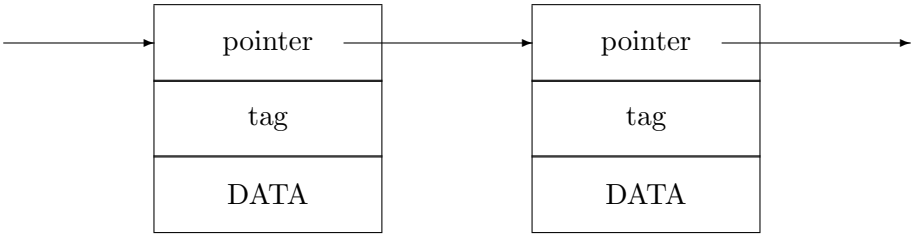
i.e. a number with about 30 decimal digits.

Hence it is necessary to use a multi-word representation, in which the maximum capacity of each word (plus 1) corresponds to the *base* or *radix*. Arithmetic in this representation must be performed word-by-word, analogously to performing decimal arithmetic by hand.

For flexibility of memory management, the words of a multi-word integer will not (generally) be contiguous as in an array, but will form a *linked list*, whose elements can each be anywhere in memory, thus:-



This means that each component or *memory cell* must be more complicated than just a word of data, and must contain both a data component and a pointer to the next cell. In practice, a list cell will probably also contain a tag component that indicates the type of data in the cell, thus:-



As a very practical side remark, the pointer might occupy one (long) word of 32 bits and the integer data component might share a second (long) word with the tag, thereby giving an integer radix of perhaps 24 bits. An integer smaller than this radix is called a *small integer*, and can be processed essentially in hardware with no list handling overhead, i.e. *fast*. REDUCE uses such small integers explicitly wherever (a) it can, and (b) speed is important, e.g. in some parts of the factorizer.

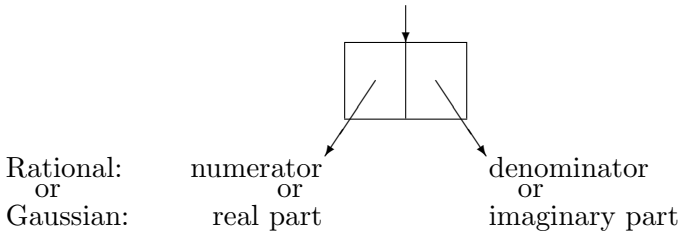
Philosophical side remark

Is the need for this non-binary arithmetic fundamental, or a consequence of current computer architectures? In particular, one could imagine an architecture without registers, in which arithmetic is performed directly in main memory, and in which bits can be randomly accessed. But the problem is that the bit-addressing overhead would grossly dominate the bit-data content. Therefore, I conjecture that it will always be desirable to manage memory in terms of groups of bits (which I will now loosely call "words").

In particular, flexible dynamic memory allocation will probably always function at the word level. Therefore, integers too large to fit into a single word will always need to be represented by a sequence of words, which implies using arithmetic with a radix > 2 (as well as binary arithmetic within each word). Binary arithmetic is particularly simple and efficient, so the larger the radix (within which binary arithmetic can be used) the better.

3.2 Other discrete number domains

Now that we have established a representation for the integers, it is easy to represent both Gaussian (i.e. complex) integers and rationals as *pairs* of integers, which in a Lisp-cell representation might look like this:-



Similarly, a complex rational is just a pair of rationals.

3.3 Rounded representation of reals

Conventional computing languages support an approximate representation of real numbers within some specified limits using a *fixed* number of bits, typically either one or two (long) words, i.e. 32 or 64 bits. In order to capture best the continuous and infinite nature of the set of real numbers the representation, called *floating-point*, uses a fixed *relative* accuracy, which corresponds to a fixed number of significant digits in a positional representation. Usually binary representation is used. A fixed number of the leading (i.e. highest order) bits are stored together with a power of 2 representing the overall magnitude, thus:-

$$R = I_1 \times 2^{I_2}.$$

Thus the representation involves two (small) integers: the maximum size of I_1 determines the acccuracy of the representation and the maximum size of I_2 determines the maximum magnitude range.

This representation is often supported by the hardware, in which case it is *fast* – probably as fast as integer arithmetic. Even if it is implemented in software it is still fairly fast because it involves only bit manipulations of one or two words.

CA systems normally support this "native" real representation (REDUCE calls it "float mode" internally), but they also support an arbitrary range, arbitrary precision version in which the two integers are the arbitrary size integers discussed above (and the exponent base is probably 10 – REDUCE calls this "bigfloat mode" internally). REDUCE (since version 3.4) switches automatically between the two representations depending on the required accuracy. Hence rounded real representation is essentially *either* a single machine float *or* a pair of integers (again)!

[Note that in practice it is necessary to tag the various possible number-pair representations to indicate what kinds of numbers they represent.]

3.4 Scalar expression representation

An algebraic expression is a heterogeneous object (its components are not all of the same type) and so an appropriately flexible data structure is necessary to represent it. Notice that even the nature of a fixed component of a polynomial is not fixed, because we want to allow polynomial coefficients to be numbers from *any* of the domains considered above or perhaps purely symbolic, and we want to allow polynomial variables to be either identifiers (variable names) or functional forms. An array would be too rigid. I will consider one possibility, a representation based on a (simply) linked list, esentially as used by REDUCE.

In this representation it is necessary only to establish a convention that fixes the meaning of each *pointer* in the structure, and the type of data pointed to need not be rigidly fixed. The physical pointer structure should match the mathematical data structure.

The structure of a *univariate polynomial* can be defined as follows, where "|" separates alternatives:-

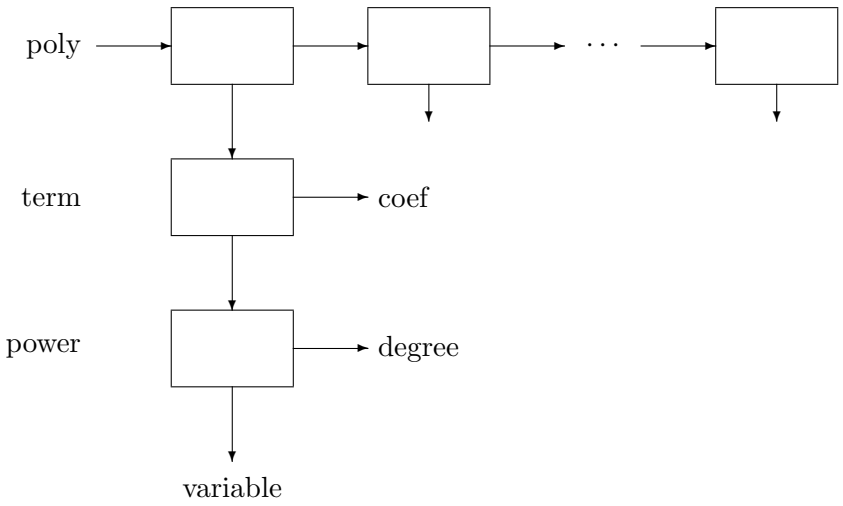
$$\begin{aligned} \text{poly} &= \text{term} + \text{poly} \mid \text{number} \mid \text{nothing} \\ \text{term} &= \text{coef} \times \text{power} \\ \text{power} &= \text{variable}^{\text{degree}} \\ \text{degree} &= \text{positive integer} \end{aligned}$$

where "number" is any element of any allowed number domain, and

$$\begin{aligned} \text{coef} &= \text{number} \\ \text{variable} &= \text{identifier} \end{aligned}$$

Note that the definition of poly(nomial) is recursive (i.e. poly is defined partly in terms of itself), and hence it *must* have a non-recursive alternative to terminate the recursion. The alternative of "nothing" is included so that 0 (the zero polynomial) need not be explicitly represented.

The mathematical structure translates immediately into the following list structure:-



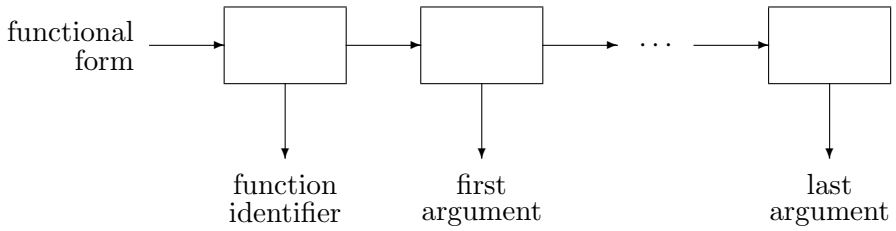
Each term has the same structure; I have shown only one term explicitly. The "· · ·" at the poly level corresponds to the recursion in the definition. This structure is a singly-linked list because there is only one path from the root (poly) to any other point, and no way back! (It is also a binary tree, since there are 2 links from each node.)

Note that this structure implies an ordering of the terms within a polynomial, indicated by the arrows at the poly level. Normally this ordering will be by decreasing degree. The components "term" and "poly" on the right of the recursive definition of poly are called respectively the *leading term* and *reductum*. The coefficient of the leading term is called the *leading coefficient*.

It is easy to replace elements of this general polynomial structure by other structures. Functional forms can be included by changing the definition of variable to

$$\text{variable} = \text{identifier} \mid \text{functional form}$$

and a reasonable list representation of a functional form is



Similarly, the definition of the polynomial coefficients can be changed to

$$\text{coef} = \text{number} \mid \text{poly}$$

which adds another form of recursion to the definition of polynomial. The polynomials appearing at different levels can involve different variables, thereby providing a representation for multivariate polynomials. Normally, each variable in a multivariate polynomial representation would appear at only one logical level in the structure. For example, this representation is equivalent to writing the following bivariate polynomial in the form

$$5x^2(y^2 + 1) + 3x(y^3 + 2y) + 7(y + 3).$$

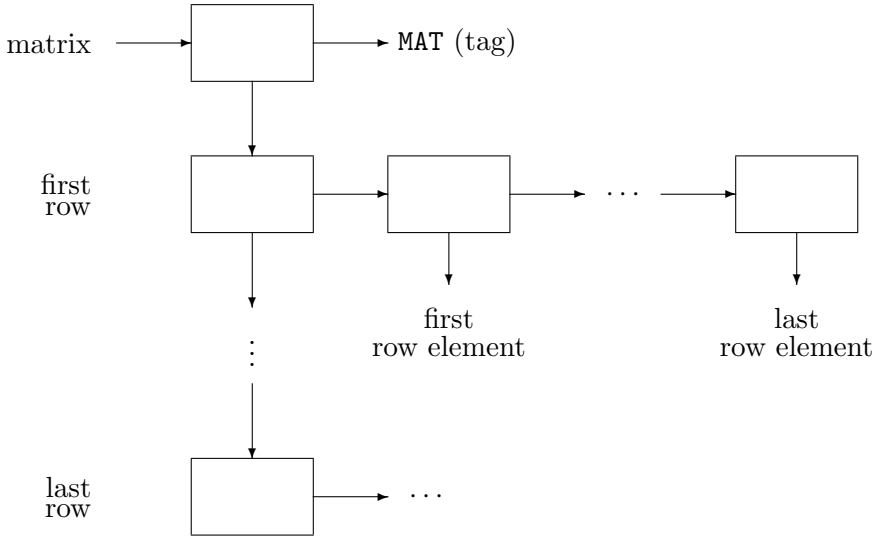
The variable appearing at the "top level" in the structure, in this example x, is called the *main variable*.

A rational function is represented as a pair of polynomials, where each polynomial may have the general form discussed above, in exactly the same way that a rational number is represented as a pair of integers.

Note that most of the representations defined so far are not unique, an important topic to which we will return later.

3.5 Representation of structured (non-scalar) data

The elements of a non-scalar data type can be represented as above. Most of the structured data types discussed earlier could reasonably be represented as either linked lists or arrays, and for matrices, vectors and tensors representation as arrays might seem most natural. However, the representation needs to carry a tag to identify its type, so the structure is not completely homogeneous even at the top level. REDUCE uses the following list representation for matrices. The structure is a list of rows preceded by a tag (the identifier `MAT`), where each row is a list of row elements, thus:-



The disadvantage of this representation is that elements cannot be randomly accessed. When REDUCE needs to access a matrix element by its indices (subscripts) it does so by stepping sequentially through the structure counting elements as it goes! But fortunately many matrix operations can be performed sequentially on the elements.

4 Sparse and dense representations

This topic concerns the representation of elements of a data structure that are zero – whether they should be represented explicitly or simply omitted. It applies to structures such as matrices and polynomials, and can be important in numerical computations with large matrices, but I will consider only polynomials here.

The degree of a polynomial is the maximum of the degrees of its individual terms, and a constant term is considered to have degree zero. (Occasionally it is necessary to define the degree of the special constant zero to be $-\infty$, but for present purposes of representation we will regard zero the same as any other constant, and give it a degree of zero.) A *dense* polynomial of a particular degree is one in which all possible terms are present, i.e. the coefficients of all terms of degree not greater than the degree of the polynomial are non-zero; if a polynomial is not dense then it is *sparse*. However, in common usage dense would be taken to mean that *most* of the possible terms were present, whereas sparse would be taken to mean that very few

of the possible terms were present.

A sparse *representation* of a polynomial is one in which only the terms with non-zero coefficients are represented explicitly, as in conventional mathematical notation, whereas a dense representation is one in which all terms with degree less than or equal to that of the polynomial are represented explicitly, e.g.

sparse: $25x^5 + 1$
dense: $25x^5 + 0x^4 + 0x^3 + 0x^2 + 0x + 1$

When using a sparse representation it is necessary to store for each term both its coefficient and its degree, whereas for a dense representation the degree can be inferred from structural information. For example, the above dense representation could be encoded simply as the list of coefficients

```
{1, 0, 0, 0, 0, 25}
```

where the position of the coefficient in the list, numbered from position 0 at its head, implicitly encodes the degree of the term. By contrast, a sparse representation would need to be encoded as a list of coefficient-degree pairs, such as

```
{ {25, 5}, {1, 0} }
```

Note that now the ordering of the pairs within the list is irrelevant (at least as far as ambiguity is concerned, but see the discussion of canonical representations below, although the ordering of the elements of each pair is still significant). Of course, it is also necessary to store the name of the polynomial variable somewhere, but this has no fundamental bearing on the distinction between sparse and dense representation.

I have illustrated the above discussion with a univariate polynomial (i.e. one with a single unknown), but it applies also to multivariate polynomials, although then as we have already seen the details of possible representations become more complicated. A sparse or dense representation is obviously most efficient respectively for a sparse or dense polynomial. In practice, for univariate polynomials it usually makes little difference, but multivariate polynomials tend to be sparse, which therefore favours a sparse representation as the primary representation in a computer algebra system that will be mainly handling multivariate polynomials. For some algebraic algorithms one or other representation is significantly more efficient, but for others it makes little difference.

REDUCE normally uses a sparse representation, for which the list representation presented above is ideally suited, although it uses a different dense representation in some special situations such as for Gröbner basis computations (to be discussed in the second part of this course). A dense representation is also called *distributed* or *distributive*.

5 Normal and canonical representations; simplification

I remarked that the representations presented earlier were not defined so as to be unique. For example, the polynomial representations could be either dense or sparse, and we did not define the term ordering to be used. The latter ambiguity corresponds to the two *textual* representations

$$x + 1 \quad \text{and} \quad 1 + x.$$

Although these two expressions are mathematically equal, their textual representations as above are different.

A representation is called *canonical* if only one representation is used for all mathematically equal expressions. The process of converting a given expression into the correct internal representation is called *simplification* and is performed by a *simplifier*. Theoretically, a simplifier is a function or mapping; in a computer algebra system a simplifier is a section of code (probably a large section consisting of a hierarchy of simplification procedures). Simplification in REDUCE is performed automatically under the control of global *switches*, whereas in most other CA systems it is not and the user must explicitly call the desired simplification functions.

The advantage of using a canonical representation is that two expressions can be compared for equality by simply comparing the data structures that represent them, without using any mathematics at all. However, it is not possible to design a canonical simplifier in all cases of interest: for example, the infamous "constants problem" of how to check for algebraic relations among transcendental field extensions remains unsolved. [For further details see Davenport *et al.* and the first chapter in Buchberger *et al.*]

A weaker, but still very useful, representation is one that is not necessarily canonical, but in which an expression that is equal to zero is always represented canonically. Such a representation and a simplifier producing it are called *normal*. Obviously, a canonical representation is necessarily normal, but not vice versa.

[Referring forward to the mathematical structures to be described next week, it is necessary to be working in an algebra that is (at least) an additive monoid (which in practice in CA one always is) in order to be able to define a normal representation. In such a monoid the only distinguished element is the zero and, moreover, in a ring or field it is not permissible to divide by zero, so it is always necessary to be able to reliably test whether a value is zero. A normal representation on an additive group gives an algorithm for testing equality by subtracting, so a canonical representation is not necessary, although it allows more efficient equality testing.]

Simplifying the result of an algebraic calculation is an essential part of the calculation; in fact, the calculation *is* the simplification of the data structure built using the various operators and functions as "constructors". One essential aim of simplification is to make full use of the unambiguous rules for performing arithmetic on numbers. By this, I mean that a simplification of the form

$$2 + 3 \rightarrow 5$$

should always be performed. This implies that polynomial terms with the same degrees should be grouped together and the resulting coefficients simplified, and any operation called simplification can be expected to do this. This operation alone, together with the requirement that a polynomial all of whose coefficients are zero is represented by the number 0, is sufficient to give a normal simplifier on univariate polynomials over the integers, which is probably the simplest non-trivial domain of algebraic objects. This is because a polynomial is zero if and only if all its coefficients are zero. The nature of a polynomial simplifier will depend on whether a sparse or dense representation is used. If a sparse representation is used then the zero polynomial is automatically represented canonically (as nothing).

Note that the distinction between nothing and the number 0 can be important. REDUCE represents the zero polynomial as the Lisp value `NIL`, which is distinct from the number 0, and it is necessary for the simplifier to convert between `NIL` and 0 when appropriate, depending on whether the underlying domain is considered to be numbers or polynomials.

To make a polynomial representation canonical it is necessary to specify a dense or sparse representation (we will assume sparse unless dense is explicitly stated), and to sort the terms using a specified ordering, as will be discussed in the next section. It is also necessary to collect terms in some way, the simplest being to multiplying out all products and powers of *expressions*, which can then either be left ungrouped or regrouped into

the recursive multivariate representation used by REDUCE. Both representations are canonical. However, this expanded representation may be very inefficient or inconvenient – for example, $(x+1)^{10}$ is a very much more succinct representation of a degree-10 polynomial than is its expanded form, and it may be convenient to perform some parts of a calculation using a non-canonical representation. An important alternative polynomial representation (which is provided by REDUCE, for example) is in a form that is fully factorized over the (possibly Gaussian or modular) integers, which is also a canonical representation.

The canonical representation for a rational *number* is a pair of integers, called the *numerator* and *denominator*, such that the denominator is strictly positive and the numerator and denominator have no common factor. If integers and rationals are to be mixed in one canonical representation then it is necessary to either give all integers an explicit denominator of 1, or to replace all rational representations having a denominator of 1 by integer representations. REDUCE does the latter.

The generalization of this to a canonical representation for rational *expressions* is as a pair of polynomials, such that the *leading term* of the denominator, in whatever polynomial representation is in use, is strictly positive, and again any common factor is cancelled between numerator and denominator. REDUCE always ensures that the denominator is "positive", but may not cancel all common factors (unless the switch `GCD` is turned on, which *by default it is not*). Hence, by default the REDUCE representation of rational expressions is not strictly canonical, although it is normal, because clearly if any factor of the numerator vanishes then so does the whole numerator and every value trivially divides 0 so that the result simplifies to 0/1, which is then simplified to the zero polynomial. The canonical representation for an expression involving sums (or differences) of rational expressions (and numbers) puts them over a common denominator, and thereby produces a single rational expression.

Simplification of multivalued functions (of which the simplest example is probably square roots) is a very difficult problem domain that has not been satisfactorily resolved.

6 Term ordering: lexicographic, total degree

Beware that the literature on this subject is at best confusing, and at worst inconsistent!

Whilst a CA system must choose a fixed polynomial term ordering to use at any particular time, it clearly could be either ascending or descending, which proves by counter-example that generally there is no unique choice of canonical representation. The terms of a univariate polynomial can be ordered only by their degree, and usually descending degree order is used because it is found to be more convenient.

For multinomials (multivariate polynomials) is is not sufficient to order terms by degree because there is no longer a unique degree, and they must also be ordered by the variables in them – for example, it must be possible to distinguish between x^2y and xy^2. Such terms have a degree with respect to each variable, and also a *total degree*, which is the sum of the degrees with respect to each variable.

The most obvious variable ordering is alphabetic. It is conventional to sort "a" before "z", and therefore to regard "a" as "more principal" than "z", which implies the ordering

$$a > b > c > \ldots > x > y > z.$$

(If non-alphabetic characters are allowed then this ordering must be generalized, e.g. to follow the character encoding scheme, which on microcomputers is normally ASCII. Generalised alphabetic ordering then implies ordering by *inverse* ASCII code.) If multi-character variable identifiers are used, as is common (and generally good practice) in computing, then the variable ordering can be on each successive character of the identifier, as used in a dictionary. CA systems have a default variable ordering, but it is normally possible to specify any desired ordering of particular variables instead, which would probably also order them before all default-ordered variables. We will assume some fixed ordering of variable identifiers – it does not matter precisely what it is.

It is now necessary to decide which ordering to apply first to terms: ordering first by variable identifier and second by degree is usually called *lexicographic ordering*; ordering first by total degree and secondly by variable identifier is usually called *total degree ordering*. Here is an example

$$\begin{array}{rl} \textbf{lexicographic:} & x^2 + 2xy + x + y^2 + y + 1 \\ \textbf{total degree:} & (x^2 + 2xy + y^2) + (x + y) + 1, \end{array}$$

where I have used alphabetic variable ordering. Note that it is conventional to write polynomial terms in *decreasing* order.

Lexicographic ordering is so called because it corresponds (once again) to dictionary ordering in which the juxtaposition within a "word" implies multiplication, powers are written as repeated products, and a missing variable is equivalent to a factor of 1 and sorts after all variables, thus

$$a > b > c > \ldots > x > y > z > 1 \text{ (meaning no variable).}$$

This ordering then implies (by applying it recursively) that

$$xx > xy > x > yy > y > 1,$$

exactly as in a dictionary (apart from the 1) and exactly as in the lexicographically ordered polynomial above.

The term "inverse lexicographic" is also used in the literature, which simply means order lexicographically and *then* reverse. For a univariate polynomial this just sorts by *increasing* degree. For the above bivariate polynomial it gives

$$\begin{array}{rl} \textbf{inverse lexicographic:} & 1 + y + y^2 + x + 2xy + x^2 \\ \textbf{total degree then inv lex:} & (y^2 + 2xy + x^2) + (y + x) + 1. \end{array}$$

Inverse lexicographic ordering alone is not very convenient because it involves increasing degree as for univariate polynomials, and total degree then inv lex is identical to total degree then lex with the variable ordering $y > x$. However, for polynomials with three or more variables total degree then inverse lexicographic ordering is not related to total degree then lexicographic ordering by simply changing the variable-name part of the ordering, and can have some advantages. For example, the homogeneous polynomial $(x + y + z)^3$ when ordered using the lexicographic ordering $x > y > z$ (the ordering of 1 is irrelevant here) has the form

$$x^3 + 3x^2y + 3x^2z + 3xy^2 + 6xyz + 3xz^2 + y^3 + 3y^2z + 3yz^2 + z^3,$$

whereas using the lexicographic ordering $z > y > x$ and *then* reversing gives the form

$$x^3 + 3x^2y + 3xy^2 + y^3 + 3x^2z + 6xyz + 3y^2z + 3xz^2 + 3yz^2 + z^3,$$

which is different!

The details of polynomial representations become particularly important when considering algorithms such as the Buchberger algorithm for constructing a Gröbner basis for a set of polynomials, when the precise choice of term

ordering can have a huge effect on the running time of the algorithm. However, in general it is not known how to choose the best ordering. Note that frequently in recent literature (and in the REDUCE Gröbner basis package) "lexicographic" is abbreviated to *lex*, "total degree then lexicographic" is referred to as *gradlex*, and "total degree then inverse lexicographic" (as originally used by Buchberger) is referred to as *revgradlex*.

The lexicographic ordering corresponds closely to the recursive representation of multivariate polynomials that we discussed earlier, because it corresponds to picking each variable in turn as a main variable and writing the polynomial as a univariate polynomial with respect to that variable. If the terms of the above bivariate polynomial example are grouped, without changing their order, it has the recursive form

$$x^2 + (2y + 1)x + (y^2 + y + 1),$$

i.e. a polynomial in x whose coefficients are polynomials in y. This might be called a *lexicographic recursive* representation.

Functional forms can easily be incorporated into the ordering scheme, for example, by ordering them by function identifier, and ordering all functional forms before all simple variables (as in REDUCE).

7 Introduction to complexity of algorithms

An *algorithm* is a description of a sequence of operations that are necessary to perform some task. Usually, one algorithm can perform a class of related tasks, where each task in the class is specified by some set of data or *parameters* – for example, a single algorithm might add any pair of integers, where the two integers themselves are the parameters. The number of operations performed by an algorithm is related to the time that will be required, and is therefore called the *time complexity* of the algorithm. The amount of intermediate data generated internally by an algorithm is related to the amount of space that will be required, and is therefore called the *space complexity* of the algorithm. Generally, these complexities depend on the parameters. Unqualified use of the term "complexity" implies the time complexity, because this is usually more significant.

There is usually more than one algorithm to perform a particular task, in which case an analysis of complexities gives some guidance as to which algorithm is better. However, the theoretical complexity of an algorithm is

not the same as its actual running time or memory requirement when implemented on a particular computer, for several reasons. Different implementations of the same algorithm on the same computer can differ significantly in their performance (depending, for example, on the implementation language used, the competence of the programmer, etc), so let us assume the best possible implementation. Different computers run at very different speeds, so absolute running times are not interesting when comparing algorithms or implementations. More significantly, different computers perform different operations at (relatively) different speeds; for example, one processor may take (relatively) much longer than another to multiply rather than to add, or to perform floating-point rather than integer arithmetic (especially if it does not have floating-point hardware support).

When performing theoretical time-complexity analysis of an algorithm it is usually assumed (probably without explicitly being stated) that there is a set of basic operations that all take the same length of time to perform, and the analysis is in terms of the number of these basic operations. For example, it might be decided that integer addition and subtraction take relatively no time at all, and that integer multiplication and division take about the same (much longer) time, and so the complexity of a problem involving only integer arithmetic might be given as the total number of multiplications and divisions, without distinguishing them. These basic integer operations would normally be for single-word integers, in terms of which the complexity of performing arithmetic on the arbitrary size multi-word integer representations that we discussed above can be analyzed, as we will do later in this course.

Even when a set of basic operations are considered all to take equivalent lengths of time, analysing the complexity of a non-trivial algorithm can still be very difficult, and there are many important algorithms for which the complexity has not been fully analysed. Usually, therefore, the *worst case* is analysed, which sets an upper bound on the complexity, although more detailed statistical analyses could be performed. Moreover, the analysis is usually *asymptotic* with respect to the parameters, meaning that it gives only the general behaviour of the complexity with respect to the complexity or size of the input data as the latter becomes infinitely large. This is reasonable because the complexity only really matters when it is large. Such asymptotic analysis discards constants of proportionality, on the grounds that an^2 is infinitely greater than bn as $n \to \infty$ regardless of the values of a and b, so the latter are not interesting. However, for small n the values of a and b can be very significant, and it may well be that problems of practical

interest do not involve very large values of n.

Hence, whilst theoretical complexity analysis is useful, it must be interpreted with care, and is not a substitute for careful experimental analysis of a careful implementation, which may give a very different picture. It is (partly) for this reason that the algorithms that have the best theoretical asymptotic behaviour are not always those that are actually implemented in CA systems! The complexity of an algorithm is also referred to as its cost, because they are related if one is actually paying for computing facilities!

Let us finish this section by putting the above remarks about asymptotic analysis on a mathematical basis. The symbol "O", meaning "of the order of", is used a lot in asymptotics, and roughly means "take the dominant behaviour as something tends to infinity, and ignore constants". More precisely, let f and g be real-valued functions defined on some ordered set S. Then we say that

- f is *dominated* by g, written $f \preceq g$ or $f = O(g)$, if there exists a positive real number c and $y \in S$ such that $|f(x)| \leq c|g(x)|$ for all $x \in S, x \geq y$, or
- f and g are *codominant*, written $f \sim g$, if both $f \preceq g$ and $g \preceq f$.

As an example, suppose an algorithm requires k operations, each of which is executed in time $c_i n^{m_i}, 1 \leq i \leq k$, where n is some parameter such as the size of the input data. Then the time complexity of the whole algorithm is $O(n^m)$, where m is the maximum of the $m_i, 1 \leq i \leq k$, (i.e. $m = \max_{i=1}^{k} m_i$).

If the time complexity of an algorithm is dominated by some fixed power of the input data size n (as in the above example), which implies that the actual complexity is (at worst) a polynomial function of n, then it is said to be a *polynomial-time* algorithm. Otherwise it is said to be an *exponential-time* algorithm, meaning that it has a complexity of the form $O(m^n)$ for some positive real number m. A problem is "easy" if it can be solved using a polynomial-time algorithm, whereas the time required for an exponential-time algorithm increases so rapidly with the input data size that relatively small problems can be unsolvable in practice, and will remain so! Note that

$$O(\log n) \prec O(n) \prec O(n \log n) \prec O(n^2) \prec O(m^n) \forall m,$$

where here "$\prec$" means "is strictly dominated by".

8 Exercises

The order of the questions follows that of the notes. Each week, answers to the question(s) marked "(** **Assessed** **)" should be handed in to me by the end of the week *after* the week in which the notes were handed out.

1. Check the estimate of ${}^{100}C_{50}$ given in the notes by computing it either exactly or at least more accurately. It is trivial to compute it using a CA system (or Lisp). In a conventional language it cannot be computed using a standard (32-bit) integer representation (why?), but it should be possible to compute it using a single-precision (32-bit) real representation, although some care will be needed in the algorithm. You may also be able to do it on a calculator.

2. Give the complete general (recursive) definition of a scalar expression, by combining the "syntax" rules given in the notes on scalar expression representation.

3. (** **Assessed** **)
Draw structure diagrams to show how the following expressions can be represented using the full sparse recursive list representation for general rational expressions described in the notes:

1. $\frac{1}{2}x^2 + \frac{3}{2}x + 1$ (as a polynomial with rational coefficients);
2. $\frac{x^2+3x+2}{2}$ (as a rational expression with integer coefficients);
3. $(y+1)x^2 + (3y)x + 2$ as a multivariate polynomial with x as main variable;
4. $\frac{x+1}{y+1}$.

4. Describe the simplest possible list representation for vectors, and outline an algorithm for adding two vectors in this representation that is explicitly sequential, i.e. steps sequentially through the list structure. Using this as a sub-algorithm, outline an algorithm for adding two matrices in the representation described in the notes, which again is explicitly sequential. [You may assume that the objects being added are conformable, i.e. have the same size.]

5. (** **Assessed** **)
A *trigonometric rational expression* in x may be defined to be a quotient of two trigonometric polynomials in x, and a *trigonometric polynomial* in x may be defined as a multivariate polynomial in which

the variables may have only the form trig(kx), where "trig" represents *any* one of the standard trigonometric functions {sin, cos, tan, csc (= cosec), sec, cot}, k may be *any* integer, and x may be any *single fixed* variable identifier. For example,

$$\frac{2\sin(2x) + 3\tan(x)}{\cos^3(3x)}.$$

Specify a canonical representation (or simplification algorithm) for such trigonometric rational expressions, assuming that there already exists a canonical representation for general rational expressions, as described in the notes. The point is to take account of all the trigonometric identities that permit different representations of expressions that are equivalent, such as

$$\cos(2x) = 2\cos^2(x) - 1 = 1 - 2\sin^2(x).$$

6. Write the polynomial $(x + y + 1)^2$ in distributed (i.e. fully expanded) form using each of the term orderings:

1. lexicographic (then degree);
2. total degree (then lexicographic).

In both cases, use the variable ordering x before y, i.e. $x > y$, and write the terms in decreasing order.

If you are feeling really enthusiastic, try writing out $(x + y + z + 1)^2$ and more complicated polynomials in lexicographic, total degree, and inverse lexicographic orderings. [You are allowed to use a CA system, although you should then at least check that you understand the result. For example, REDUCE 3.4 supports all three orderings via its `GROEBNER` package, which will need to be explicitly loaded ("`LOAD_PACKAGE GROEBNER;`"). Then select a term ordering ("`TORDER LEX;`", the default, "`TORDER GRADLEX;`" or "`TORDER REVGRADLEX;`"), and compute the Gröbner basis of a single polynomial ("`GROEBNER {POLY};`"). Derive (2.02) appears to offer only lexicographic ordering, although one can choose the variable ordering.]

7. Give a precise meaning to the assertion that

$$O(\log n) \prec O(n) \prec O(n \log n) \prec O(n^2) \prec O(m^n) \forall m,$$

and then prove the assertion.