Symbolic Computation 3:

NORMAL FORMS AND DATA STRUCTURES

by

Keith O. Geddes
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

Research Report CS-81-29

September 1981

PREFACE

This report consists of Chapter 3 of a textbook being written under the title 'Algebraic Algorithms for Symbolic Computation' by Keith O. Geddes. The table of contents for Chapter 3 appears below. A more complete table of contents for the textbook appears on the following pages.

Chapter 3:	NORMAL FORMS AND DATA STRUCTURES	3-1
3.1.	Levels of Abstraction	3-1
3.2.	Normal Form and Canonical Form	3-2
3.3.	Normal Forms for Polynomials	3-5
3.4.	Normal Forms for Rational Functions and Power Series	3-8
3.5.	Data Structures for Multiprecision Integers and Rational Numbers	3-12
3.6.	Data Structures for Polynomials, Rational Functions, and Power Series	3-15
	Bibliography	3-22
	Exercises	3-23

ALGEBRAIC ALGORITHMS FOR SYMBOLIC COMPUTATION

KEITH O. GEDDES

Department of Computer Science
University of Waterloo

CONTENTS

Chapter 1:	INTRODUCTION	
1.1.	What is Symbolic Computation?	
1.2.	A Brief Historical Sketch	
1.3.	Algorithmic Notation	
1.4.	Analysis of Algorithms	
	Bibliography	
	Exercises	
Chapter 2:	ALGEBRA OF POLYNOMIALS, RATIONAL FUNCTIONS, AND POWER SERIES	2-1
2.1.	Rings and Fields	2-1
2.2.	Divisibility and Factorization in Integral Domains	2-3
2.3.	The Euclidean Algorithm	2-8
2.4.	Univariate Polynomial Domains	2-13
2.5.	Multivariate Polynomial Domains	2-19
2.6.	The Primitive PRS Euclidean Algorithm	2-24
2.7.	Quotient Fields and Rational Functions	2-30
2.8.	Power Series and Extended Power Series	2-33
2.9.	Relationships Among Domains	2-39
	Bibliography	2-41
	Exercises	2-42
Chapter 3:	NORMAL FORMS AND DATA STRUCTURES	3-1
3.1.	Levels of Abstraction	3-1
3.2.	Normal Form and Canonical Form	3-2
3.3.	Normal Forms for Polynomials	3-5
3.4.	Normal Forms for Rational Functions and Power Series	3-8
3.5.	Data Structures for Multiprecision Integers and Rational Numbers	3-12
3.6.	Data Structures for Polynomials, Rational Functions, and Power Series	3-15
	Bibliography	3-22
	Exercises	3-23
Chapter 4:	ARITHMETIC ON POLYNOMIALS, RATIONAL FUNCTIONS, AND POWER SERIES	4-1
4.1.	Arithmetic in the Finite Field $\mathbf{Z}_p$	
4.2.	Arithmetic on Multiprecision Integers	
4.3.	Arithmetic on Polynomials and Rational Functions	

4.4.	Arithmetic on Power Series	
Chapter 5:	HOMOMORPHISMS AND CHINESE REMAINDER ALGORITHMS	5-1
5.1.	Ring Morphisms	5-1
5.2.	Characterization of Morphisms	5-6
5.3.	Homomorphic Images	5-12
5.4.	The Integer Chinese Remainder Algorithm	5-18
5.5.	The Polynomial Interpolation Algorithm	5-24
5.6.	Further Discussion of the Two Algorithms	5-28
	Bibliography	5-34
	Exercises	5-35
Chapter 6:	NEWTON'S ITERATION AND THE HENSEL CONSTRUCTION	6-1
6.1.	P-adic and Ideal-adic Representations	6-1
6.2.	Newton's Iterations for $f(u) = 0$	6-8
6.3.	Hensel's Lemma	
6.4.	The Univariate EZ Lifting Algorithm	
6.5.	Special Techniques for the Non-monic Case	
6.6.	The Multivariate EZ Lifting Algorithm	
Chapter 7:	POLYNOMIAL GCD COMPUTATION AND POLYNOMIAL FACTORIZATION	7-1
Chapter 8:	SOLVING EQUATIONS AND THE SIMPLIFICATION PROBLEM	8-1
Chapter 9:	SYMBOLIC INTEGRATION	9-1

3. NORMAL FORMS AND DATA STRUCTURES

This chapter is concerned with the computer representation of the algebraic objects discussed in chapter 2. The zero equivalence problem is introduced and the important concepts of normal form and canonical form are defined. Various normal forms are presented for polynomials, rational functions, and power series. Finally data structures are considered for the representation of multiprecision integers, rational numbers, polynomials, rational functions, and power series.

3.1. LEVELS OF ABSTRACTION

In chapter 2 we discussed domains of polynomials, rational functions, and power series in an abstract setting. That is to say, a polynomial (for example) was considered to be a basic object in a domain D[x] in the same sense that an integer is considered to be a basic object when discussing the properties of the domain **Z**. The ring operations of + and × were considered in chapter 2 to be primitive operations on the objects in the domain under consideration. However when we consider a computer implementation for representing and manipulating the objects in these various domains, we find that there is a great difference in the complexity of the data structures required to represent a polynomial, rational function, or power series compared with the representation of an integer. Also, at the machine level the complexity of the algorithms defining the ring operations is very dependent on the actual objects being manipulated.

While the point of view used in chapter 2 is too abstract for purposes of understanding issues such as the complexity of polynomial multiplication, the data structure level (where a distinction is made, for example, between a linked list representation and an array representation) is too low-level for convenience. It is useful to consider an intermediate level of abstraction between these two extremes. Three levels of abstraction will be identified as follows:

(i) The *object* level is the abstract level where the elements of a domain are considered to be primitive objects.

(ii) The *form* level is the level of abstraction in which we are concerned with how an object is represented in terms of some chosen 'basic symbols', recognizing that a particular object may have many different valid representations in terms of the chosen symbols. For example, at this level we would distinguish between the following different representations of the same bivariate polynomial in the domain **Z**[x,y]:

(1) $a(x,y) = 12x^2y - 4xy + 9x - 3$;

(2) $a(x,y) = (3x - 1)(4xy + 3)$;

(3) $a(x,y) = (12y)x^2 + (-4y + 9)x - 3$.

(iii) The *data structure* level is where we are concerned with the organization of computer memory used in representing an object in a particular form. For example, the polynomial $a(x,y)$ in the form (1) could be represented by a linked list consisting of four links (one for each term), or $a(x,y)$ could be represented by an array of length six containing the integers 12, 0, −4, 9, 0, −3 as the coefficients of a bivariate polynomial with implied exponent vectors (2, 1), (2, 0), (1, 1), (1, 0), (0, 1), (0, 0) in that order, or $a(x,y)$ could be represented

by some other data structure.

In a high-level computer language for symbolic computation the operations such as + and × will be used as primitive operations in the spirit of the object level of abstraction. However in succeeding chapters we will be discussing algorithms for various operations on polynomials, rational functions, and power series and these algorithms will be described at the form level of abstraction. The next three sections discuss in more detail the various issues of form which arise. Then choices of data structures for each of the above classes of objects will be considered.

3.2. NORMAL FORM AND CANONICAL FORM

The Problem of Simplification

When symbolic expressions are formed and manipulated, there soon arises the general problem of *simplification*. For example, the manipulation of bivariate polynomials might lead to the expression

$$(4) \quad (12x^2y - 4xy + 9x - 3) - (3x - 1)(4xy + 3).$$

Comparing with (1) and (2) it can be seen that the expression (4) is the zero polynomial as an object in the domain $\mathbf{Z}[x,y]$. Clearly it would be a desirable property of a system for polynomial manipulation to replace the expression (4) by the expression 0 as soon as it is encountered. There are two important aspects to this problem:

(i) a large amount of computer resources (memory space and execution time) may be wasted storing and manipulating unsimplified expressions (indeed a computation may exhaust the allocated computer resources before completion because of the space and time consumed by unsimplified expressions); and

(ii) from a human engineering point of view, we would like results to be expressed in their simplest possible form.

The problem of algorithmically specifying the 'simplest' form for a given expression is a very difficult problem. For example, when manipulating polynomials from the domain $\mathbf{Z}[x,y]$ we could demand that all polynomials be fully expanded (with like terms combined appropriately), in which case the expression (4) would be represented as the zero polynomial. However consider the expression

$$(5) \quad (x + y)^{1000} - y^{1000};$$

the expanded form of this polynomial will contain a thousand terms and from either the human engineering point of view or computer resource considerations, expression (5) would be considered 'simpler' as it stands than in expanded form. Similarly, the expression

$$(6) \quad x^{1000} - y^{1000}$$

which is in expanded form is 'simpler' than a corresponding factored form in which $(x - y)$ is factored out.

Zero Equivalence

The *zero equivalence* problem is the special case of the general simplification problem in which we are concerned with recognizing when an expression is equivalent to zero. This special case is singled out because it is a well-defined problem (whereas 'simplification' is not well-defined until an ordering is imposed on the expressions to indicate when one expression is to be considered simpler than another) and also because an algorithm for determining zero equivalence is considered to be a sufficient 'simplification' algorithm in some practical situations. However even this well-defined subproblem is a very difficult problem. For example, when manipulating the more general functions to be considered later in this book one might encounter the expression

(7) $\log \tan(\frac{x}{2} + \frac{\pi}{4}) - \sinh^{-1} \tan x,$

which can be recognized as zero only after very nontrivial transformations. A discussion of the zero equivalence problem at this level of generality will be postponed until a later chapter, where we find that in a 'sufficiently rich' class of expressions the zero equivalence problem is recursively undecidable. Fortunately though, the problem can be solved in many particular classes of expressions of practical interest. In particular the cases of polynomials, rational functions, and power series do not pose any serious difficulties.

Transformation Functions

The simplification problem can be treated in a general way as follows. Consider a set E of expressions and let $\sim$ be an equivalence relation defined on E. Then $\sim$ partitions E into equivalence classes and the quotient set $E/\sim$ denotes the set of all equivalence classes. (This terminology has already been introduced in chapter 2 for the special case where E is the set of quotients of elements from an integral domain). The simplification problem can then be treated by specifying a transformation f: E $\rightarrow$ E such that for any expression $a \in E$, the transformed expression $f(a)$ belongs to the same equivalence class as a in the quotient set $E/\sim$. Idealistically it would be desired that $f(a)$ be 'simpler' than a.

In stating the definitions and theorems in this section we will use the symbol $\equiv$ to denote the relation 'is identical to' at the form level of abstraction (i.e. identical as strings of symbols). For example, the standard mathematical use of the symbol = denotes the relation 'is equal to' at the object level of abstraction so that

$$12x^2y - 4xy + 9x - 3 = (3x - 1)(4xy + 3),$$

whereas the above relation is not true if = is replaced by $\equiv$. In fact the relation = of mathematical equality is precisely the equivalence relation $\sim$ which we have in mind here. (In future sections there will be no confusion in reverting to the more general use of = for both $\equiv$ and $\sim$ since the appropriate meaning will be clear from the context).

Definition 3.1.

Let E be a set of expressions and let $\sim$ be an equivalence relation on E. A *normal function* for [E; $\sim$] is a computable function f: E $\rightarrow$ E which satisfies the following properties:

(i) $f(a) \sim a$ for all $a \in E$;

(ii) $a \sim 0 \Rightarrow f(a) \equiv f(0)$ for all $a \in E$. □

Definition 3.2.

Let [E; $\sim$] be as above. A *canonical function* for [E; $\sim$] is a normal function f: E $\rightarrow$ E which satisfies the additional property:

(iii) $a \sim b \Rightarrow f(a) \equiv f(b)$ for all $a,b \in E$. □

Definition 3.3.

If f is a normal function for [E; $\sim$] then an expression $\tilde{a} \in E$ is said to be a *normal form* if $f(\tilde{a}) \equiv \tilde{a}$. If f is a canonical function for [E; $\sim$] then an expression $\tilde{a} \in E$ is said to be a *canonical form* if $f(\tilde{a}) \equiv \tilde{a}$. □

Example 3.1.

Let E be the domain $\mathbf{Z}[x]$ of univariate polynomials over the integers. Consider the normal functions f_1 and f_2 specified as follows:

f_1: (i) multiply out all products of polynomials;

(ii) collect terms of the same degree.

f_2: (i) multiply out all products of polynomials;

(ii) collect terms of the same degree;

(iii) rearrange the terms into descending order of their degrees.

Then f_1 is a normal function which is not a canonical function and f_2 is a canonical function. A normal form for polynomials in $\mathbf{Z}[x]$ corresponding to f_1 is

$$a_1x^{e_1} + a_2x^{e_2} + \cdots + a_mx^{e_m} \quad \text{with} \quad e_i \neq e_j \text{ when } i \neq j.$$

A canonical form for polynomials in $\mathbf{Z}[x]$ corresponding to f_2 is

$$a_1x^{e_1} + a_2x^{e_2} + \cdots + a_mx^{e_m} \text{ with } e_i < e_j \text{ when } i > j. \quad \square$$

It is obvious that in a class E of expressions for which a normal function has been defined, the zero equivalence problem is solved. However there is not a unique normal form for all expressions in a particular equivalence class in $E/\sim$ unless the equivalence class contains 0. A canonical form, in contrast, provides a unique representative for each equivalence class in $E/\sim$, as the following theorem proves.

Theorem 3.1.

If f is a canonical function for [E; $\sim$] then the following properties hold:

(i) f is idempotent (i.e. $f \circ f \equiv f$ where $\circ$ denotes composition of functions);

(ii) $f(a) \equiv f(b)$ if and only if $a \sim b$;

(iii) in each equivalence class in $E/\sim$ there exists a unique canonical form.

Proof:

(i) $f(a) \sim a$ for all $a \in E$, by Definition 3.1 (i)

$\Rightarrow$ $f(f(a)) \equiv f(a)$ for all $a \in E$, by Definition 3.2 (iii).

(ii) 'if': This holds by Definition 3.2 (iii).
'only if': Let $f(a) \equiv f(b)$. Then

$$a \sim f(a) \equiv f(b) \sim b \quad \text{by Definition 3.1 (i)}$$

$\Rightarrow$ $a \sim b$.

(iii) 'Existence': Let a be any element of a particular equivalence class. Define $\tilde{a} \equiv f(a)$. Then

$$f(\tilde{a}) \equiv f(f(a))$$
$$\equiv f(a), \text{ by idempotency}$$
$$\equiv \tilde{a}.$$

'Uniqueness': Suppose $\tilde{a}_1$ and $\tilde{a}_2$ are two canonical forms in the same equivalence class in $E/\sim$. Then

$\tilde{a}_1 \sim \tilde{a}_2$

$\Rightarrow$ $f(\tilde{a}_1) \equiv f(\tilde{a}_2)$ by Definition 3.2 (iii)

$\Rightarrow$ $\tilde{a}_1 \equiv \tilde{a}_2$ by definition of canonical form. $\square$

3.3. NORMAL FORMS FOR POLYNOMIALS

Multivariate Polynomial Representations

The problem of representing multivariate polynomials gives rise to several important issues at the form level of abstraction. One such issue was briefly encountered in chapter 2, namely the choice between recursive representation and distributive representation. In the *recursive representation* a polynomial $a(x_1, \ldots, x_v) \in D[x_1, \ldots, x_v]$ is represented as

$$a(x_1, \ldots, x_v) = \sum_{i=0}^{\partial_1[a(x)]} a_i(x_2, \ldots, x_v) x_1^i$$

(i.e. as an element of the domain $D[x_2, \ldots, x_v][x_1]$) where, recursively, the polynomial coefficients $a_i(x_2, \ldots, x_v)$ are represented as elements of the domain $D[x_3, \ldots, x_v][x_2]$, and so on so that ultimately the polynomial $a(x_1, \ldots, x_v)$ is viewed as an element of the domain $D[x_v][x_{v-1}] \cdots [x_1]$. An example of a polynomial from the domain $\mathbf{Z}[x,y,z]$ expressed in the recursive representation is:

(8) $a(x,y,z) = (3y^2 + (-2z^3)y + 5z^2)x^2 + (4)x + ((-6z + 1)y^3 + 3y^2 + (z^4 + 1)).$

In the *distributive representation* a polynomial $a(\mathbf{x}) \in D[\mathbf{x}]$ is represented as

$$a(\mathbf{x}) = \sum_{\mathbf{e} \in N^v} a_{\mathbf{e}} \mathbf{x}^{\mathbf{e}}$$

where $a_{\mathbf{e}} \in D$. For example, the polynomial $a(x,y,z) \in \mathbf{Z}[x,y,z]$ given in (8) could be expressed in the distributive representation as

(9) $a(x,y,z) = 3x^2y^2 - 2x^2yz^3 + 5x^2z^2 + 4x - 6y^3z + y^3 + 3y^2 + z^4 + 1.$

Another representation issue which arises is the question of sparse versus dense representation. This issue has to do with whether or not terms with zero coefficients are explicitly represented. In the *sparse representation* only the terms with nonzero coefficients are represented while in the *dense representation* all terms which could possibly appear in a polynomial of the specified degree are represented (whether or not some of these terms have zero coefficients in a specific case). For example, a natural representation of univariate polynomials $\sum_{i=0}^{n} a_i x^i \in \mathbf{Z}[x]$ of specified maximum degree n using arrays is to store the $(n+1)$-array $(a_0, \ldots, a_n)$; this is a dense representation since zero coefficients will be explicitly stored. While it is quite possible to generalize this example to obtain a corresponding dense representation for multivariate polynomials (e.g. by imposing the lexicographical ordering of exponent vectors), such a representation is found to be highly impractical. For if a particular computation involves the manipulation of polynomials in v indeterminates with maximum degree d in each indeterminate then the number of coefficients which must be stored for each specific polynomial is $(d+1)^v$. It is not an uncommonly large problem to have, for example, $v = 5$ and $d = 15$ in which case each polynomial requires the storage of over a million coefficients. In a practical problem with $v = 5$ and $d = 15$ most of the million coefficients will be zero, since otherwise the computation being attempted is beyond the capacity of present-day computers. (And who would care to *look* at an expression containing a million terms!) All of the major systems for symbolic computation therefore use the sparse representation for multivariate polynomials.

A similar choice must be made as to whether or not to store *zero exponents*. When we express polynomials such as (8) and (9) on the written page we do not explicitly write those monomials which have zero exponents (just as we naturally use the sparse representation of polynomials on the written page). However the penalty in memory space for choosing to store zero exponents is not excessive while such a choice may result in more efficient algorithms for manipulating polynomials. We find that in some systems zero exponents are not stored and in other systems they are stored.

The polynomial representation issues discussed so far do not address the specification of normal or canonical forms. While all of these issues can be considered at the form level of abstraction, the three issues discussed above involve concepts that are closer to the data structure level than the issue of normal/canonical forms. A hierarchy of the levels of abstraction of these various representation issues is illustrated in *Figure 3.1.*

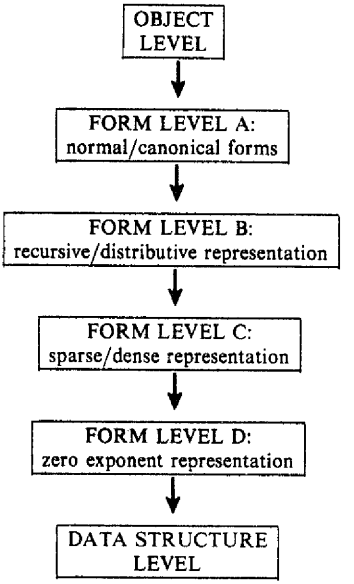


Figure 3.1. Levels of Abstraction for Multivariate Polynomial Representations

Definitions of Normal Forms

Definition 3.4.

An *expanded normal form* for polynomial expressions in a domain $D[x_1, \ldots, x_v]$ can be specified by the normal function

f_1: (i) multiply out all products of polynomials;

(ii) collect terms of the same degree.

An *expanded canonical form* for polynomial expressions in a domain $D[x_1, \ldots, x_v]$ can be specified by the canonical function

f_2: apply f_1, then

(iii) rearrange the terms into descending order of their degrees. □

Definition 3.5.

A *factored normal form* for polynomial expressions in a domain $D[x_1, \ldots, x_v]$ can be specified by the normal function

f_3: if the expression is in the product form $\prod_{i=1}^{k} p_i$, $p_i \in D[x_1, \ldots, x_v]$ for $i = 1, 2, \ldots, k$, where no p_i is itself in product form, then replace the expression by $\prod_{i=1}^{k} f_2(p_i)$ where f_2 is the canonical function defined in Definition 3.4 and where the latter product is understood to be zero if any of its factors is zero.

A *factored canonical form* for polynomial expressions in a domain $D[x_1, \ldots, x_v]$ (assuming that D is a UFD) can be specified by the canonical function

f_4: apply f_3 and if the result is nonzero then factorize each $f_2(p_i)$ into its unit normal factorization (according to Definition 2.13) and collect factors to obtain the unit normal factorization of the complete expression (made unique by imposing a pre-specified ordering on the factors). □

It should be noted that in an implementation of the transformation functions f_1 and f_2 of Definition 3.4, the concept of 'degree' means 'exponent vector' if the distributive representation is used and it means 'univariate degree' applied to each successive level of recursion if the recursive representation is used. Note also that in the specification of the transformation function f_3 of Definition 3.5, the canonical function f_2 could be replaced by the normal function f_1 and f_3 would still be a valid normal function. However the normal function f_3 as specified is more often used in practical systems. Finally note that the factors p_i $(1 \leqslant i \leqslant k)$ appearing in Definition 3.5 are not necessarily distinct (i.e. there may be some repeated factors).

Example 3.2.

The polynomial $a(x,y,z)$ in (8) is expressed in expanded canonical form using the recursive representation for the domain $\mathbf{Z}[x,y,z]$. The same polynomial is expressed in (9) in expanded canonical form using the distributive representation for the domain $\mathbf{Z}[x,y,z]$. □

Example 3.3.

Let $a(x,y) \in \mathbf{Z}[x,y]$ be the expression

$$a(x,y) = ((x^2 - xy + x) + (x^2 + 3)(x - y + 1))((y^3 - 3y^2 - 9y - 5) + x^4(y^2 + 2y + 1)).$$

Using the distributive representation for writing polynomials, an expanded normal form obtained by applying f_1 of Definition 3.4 to $a(x,y)$ might be (depending on the order in which the multiplication algorithm produces the terms):

$$\begin{aligned} f_1(a(x,y)) = {} & 5x^2y^3 + 3x^2y^2 - 13x^2y - 10x^2 + 3x^6y + 2x^6 - xy^4 + 7xy^3 - 3xy^2 - 31xy \\ & - x^5y^3 + 2x^5y^2 + 7x^5y - 20x + 4x^5 + x^3y^3 - 3x^3y^2 - 9x^3y - 5x^3 + x^7y^2 + 2x^7y \\ & + x^7 - x^2y^4 - x^6y^3 - 3y^4 + 12y^3 + 18y^2 - 12y - 3x^4y^3 - 3x^4y^2 + 3x^4y - 15 + 3x^4. \end{aligned}$$

The expanded canonical form obtained by applying f_2 of Definition 3.4 to $a(x,y)$ is

$$\begin{aligned} f_2(a(x,y)) = {} & x^7y^2 + 2x^7y + x^7 - x^6y^3 + 3x^6y + 2x^6 - x^5y^3 + 2x^5y^2 + 7x^5y + 4x^5 \\ & - 3x^4y^3 - 3x^4y^2 + 3x^4y + 3x^4 + x^3y^3 - 3x^3y^2 - 9x^3y - 5x^3 - x^2y^4 + 5x^2y^3 + 3x^2y^2 \\ & - 13x^2y - 10x^2 - xy^4 + 7xy^3 - 3xy^2 - 31xy - 20x - 3y^4 + 12y^3 + 18y^2 - 12y - 15. \end{aligned}$$

Applying, respectively, f_3 and f_4 of Definition 3.5 to $a(x,y)$ yields the factored normal form

$$f_3(a(x,y)) = (x^3 - x^2y + 2x^2 - xy + 4x - 3y + 3)(x^4y^2 + 2x^4y + x^4 + y^3 - 3y^2 - 9y - 5)$$

and the factored canonical form

$$f_4(a(x,y)) = (x - y + 1)(x^2 + x + 3)(x^4 + y - 5)(y + 1)^2. \quad \square$$

Some Practical Observations

The normal and canonical functions f_1, f_2, f_3, and f_4 of Definitions 3.4-3.5 are not all practical functions to implement in a system for symbolic computation. Specifically the canonical function f_4 (factored canonical form) is rarely used in a practical system because polynomial factorization is a relatively expensive operation (see chapter 8). On the other hand, one probably would not choose to implement the normal function f_1 since the canonical function f_2 requires little additional cost and yields a canonical (unique) form. One therefore finds in several systems for symbolic computation a variation of the canonical function f_2 (expanded canonical form) and also a variation of the normal function f_3 (factored normal form), usually with some form of user control over which 'simplification' function is to be used for a given computation. It can be seen from Example 3.3 that the normal function f_3 might sometimes be preferable to the canonical function f_2 because it may leave the expression in a more compact form, thus saving space and also possibly saving execution time in subsequent operations. On the other hand, the canonical function f_2 would 'simplify' the expression

$$\begin{aligned} a(x,y) = (x - y)(&x^{19} + x^{18}y + x^{17}y^2 + x^{16}y^3 + x^{15}y^4 + x^{14}y^5 + x^{13}y^6 + x^{12}y^7 \\ &+ x^{11}y^8 + x^{10}y^9 + x^9y^{10} + x^8y^{11} + x^7y^{12} + x^6y^{13} + x^5y^{14} + x^4y^{15} + x^3y^{16} \\ &+ x^2y^{17} + xy^{18} + y^{19}) \end{aligned}$$

into the expression

$$f_2(a(x,y)) = x^{20} - y^{20}$$

while the normal function f_3 would leave $a(x,y)$ unchanged. Finally it should be noted that both f_2 and f_3 would transform the expression (5) into an expression containing a thousand terms and therefore it is desirable to also have in a system a weaker 'simplifying' function (e.g. MACSYMA's 'general simplifier') which would not apply any transformation to an expression like (5). The latter type of transformation function would be neither a canonical function nor a normal function.

3.4. NORMAL FORMS FOR RATIONAL FUNCTIONS AND POWER SERIES

Rational Functions

Recall that a field $D(x_1, \ldots, x_v)$ of rational functions is simply the quotient field of a polynomial domain $D[x_1, \ldots, x_v]$. The choice of normal forms for rational functions therefore follows quite naturally from the polynomial forms that are chosen. The general concept of a canonical form for elements in a quotient field was defined in section 2.7 by conditions (2.75)-(2.77), which becomes the following definition for the case of rational functions if we choose the expanded canonical form of Definition 3.4 for the underlying polynomial domain. (We will assume that D is a UFD so that GCD's exist).

Definition 3.6.

An *expanded canonical form* for rational expressions in a field $D(x_1, \ldots, x_v)$ can be specified by the canonical function

f_5: (i) [form common denominator] put the expression into the form a/b where $a,b \in D[x_1, \ldots, x_v]$ by performing the arithmetic operations according to equations (2.73)-(2.74);

(ii) [satisfy condition (2.75): remove GCD] compute $g = \mathrm{GCD}(a,b) \in \mathrm{D}[x_1, \ldots, x_v]$ (e.g. by using Algorithm 2.3) and replace the expression a/b by a'/b' where $a = a'g$ and $b = b'g$;

(iii) [satisfy condition (2.76): unit normalize] replace the expression a'/b' by a''/b'' where $a'' = a'[\mathrm{u}(b')]^{-1}$ and $b'' = b'[\mathrm{u}(b')]^{-1}$;

(iv) [satisfy condition (2.77): make polynomials canonical] replace the expression a''/b'' by $\mathrm{f}_2(a'')/\mathrm{f}_2(b'')$ where f_2 is the canonical function of Definition 3.4. □

It is not made explicit in the above definition of canonical function f_5 whether or not some normal or canonical function would be applied to the numerator and denominator polynomials (a and b) computed in step (i). It might seem that in order to apply Algorithm 2.3 in step (ii) the polynomials a and b need to be in expanded canonical form, but as a practical observation it should be noted that if instead a and b are put into factored normal form (for example) then step (ii) can be carried out by applying Algorithm 2.3 separately to the various factors. It will be seen in chapter 4 that the latter approach leads to a more efficient implementation.

As in the case of polynomials, it can be useful to consider non-canonical normal forms for rational functions (and indeed more general forms which are neither canonical nor normal). We will not set out formal definitions of normal forms for rational expressions but several possible normal forms can be outlined as follows:

factored/factored: numerator and denominator both in factored normal form;

factored/expanded: numerator in factored normal form and denominator in expanded canonical form;

expanded/factored: numerator in expanded canonical form and denominator in factored normal form.

In this notation the expanded canonical form of Definition 3.6 would be denoted as *expanded/expanded*. In the above we are assuming that conditions (2.75) and (2.76) are satisfied but that condition (2.77) is not necessarily satisfied. Noting that to satisfy condition (2.75) (i.e. to remove the GCD of numerator and denominator) requires a (relatively expensive) GCD computation, it can be useful to consider four more normal forms for rational functions obtained from the above four numerator/denominator combinations with the additional stipulation that condition (2.75) is not necessarily satisfied.

Among these various normal forms one that has been found to be particularly useful for the efficient manipulation of rational expressions is the *expanded/factored* normal form, with condition (2.75) satisfied by an efficient scheme for GCD computation which exploits the presence of explicit factors whenever an arithmetic operation is performed. (See chapter 4 for details). Such a choice is the default mode in ALTRAN, with the other normal forms available by user specification. Finally noting that step (i) of function f_5 (Definition 3.6) is itself nontrivial, a weaker 'simplifying' function such as MACSYMA'S 'general simplifier' chooses to leave expressions in a less transformed state yielding neither a canonical nor a normal form (until the user requests a 'rational canonicalization').

Power Series: The TPS Representation

The representation of power series poses the problem of finding a *finite* representation for an *infinite* expression. Obviously we cannot represent a power series in a form directly analogous to the expanded canonical form for polynomials because there are an infinite number of terms. One common solution to this problem is to use the *truncated power series* (TPS) representation in which a power series

$$\textbf{(10)} \quad \mathrm{a}(x) = \sum_{i=0}^{\infty} a_k x^k \quad \in \mathrm{D}[[x]]$$

is represented as

$$(11) \quad \sum_{k=0}^{t} a_k x^k$$

where t is a specified *truncation degree*. Thus only a finite number of terms are actually represented and a TPS such as (11) looks exactly like a polynomial. However a distinction must be made between a polynomial of degree t and TPS with truncation degree t because the results of arithmetic operations on the two types of objects are not identical. In order to make this distinction it is convenient to use the following notation for the TPS representation (with truncation degree t) of the power series (10):

$$(12) \quad a(x) = \sum_{k=0}^{t} a_k x^k + O(x^{t+1}),$$

where in general the expression $O(x^p)$ denotes an unspecified power series $\alpha(x)$ with $\mathrm{ord}[\alpha(x)] \geqslant p$.

The non-exact nature of the TPS representation of power series poses a problem when we consider normal forms for power series. For if in Definition 3.1 we consider the set E of expressions to be the set of all (infinite) power series then the transformation performed by representing a power series in its TPS representation (with specified truncation degree t) violates the first property of a normal function. Specifically, two power series which are not equivalent will be transformed into the same TPS representation if they happen to have identical coefficients through degree t. On the other hand, we can take a more practical point of view and consider the set E_t of all TPS expressions of the form (12) with specified truncation degree t. Since we are only considering univariate power series domains with the most general coefficient domain being a field of rational functions, it follows immediately that (12) is a normal form for the set E_t if we choose a normal form for the coefficients and it is a canonical form for the set E_t if we choose a canonical form for the coefficients.

Power Series: Non-Truncated Representations

The TPS representation is not the only approach by which an infinite power series can be finitely represented. There are representations which are both finite and *exact*. For example, the Taylor series expansion about the point $x = 0$ of the function e^x might be written as

$$(13) \quad \sum_{k=0}^{\infty} \frac{1}{k!} x^k$$

which is an exact representation of the complete infinite series using only a finite number of symbols. The form (13) is a special instance of what we shall name the *non-truncated power series* (NTPS) representation of a power series $a(x) \in D[[x]]$ which takes the general form

$$(14) \quad a(x) = \sum_{k=0}^{\infty} f_a(k) x^k,$$

where $f_a(k)$ is a specified *coefficient function* (defined for all nonnegative integers k) which computes the k-th coefficient. By representing the coefficient function $f_a(k)$, the infinite power series $a(x)$ is fully represented. The fact that only a finite number of coefficients could ever be explicitly *computed* is in this way separated from the *representation* issue and, unlike the TPS representation, there is no need to pre-specify the maximum number of coefficients which may eventually be explicitly computed.

If $a(x)$ is the power series (13) then the coefficient function can be specified by

$$(15) \quad f_a(k) ::= \frac{1}{k!}.$$

In a practical implementation of the NTPS representation it would be wise to store coefficients

that are explicitly computed so that they need not be re-computed when and if they are required again later in a computation. Thus at a particular point in a computation if the first l coefficients for $a(x)$ have previously been explicitly computed and stored in a linear list $\mathbf{a} = (a_0, a_1, \ldots, a_{l-1})$ then the specification of the coefficient function should be changed from (15) to

(16) $f_a(k) ::=$ **if** $k < l$ **then** a[k] **else** $\frac{1}{k!}$

where a[k] denotes an element access in the linear list **a**. Initially $l = 0$ in specification (16) and in general l is the current *length* of **a**. It can also be seen that from the point of view of computational efficiency it might be better to change specification (16) to

(17) $f_a(k) ::=$ **if** $k = 0$ **then** 1 **else if** $k < l$ **then** a[k] **else** $\frac{f_a(k-1)}{k}$

where the recurrence $a_k = \frac{a_{k-1}}{k}$ (for $k > 0$) will be used to compute successive coefficients.

The specification of the coefficient function can become even more complex than indicated above. For example if $a(x)$ and $b(x)$ are two power series with coefficient functions $f_a(k)$ and $f_b(k)$ then the sum

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

can be specified by the coefficient function

(18) $f_c(k) ::= f_a(k) + f_b(k)$

and the product

$$d(x) = a(x)\,b(x)$$

can be specified by the coefficient function

(19) $f_d(k) ::= \sum_{i=0}^{k} f_a(i)\, f_b(k-i).$

If $f_a(k)$ and $f_b(k)$ are explicit expressions in k then the coefficient function $f_c(k)$ in (18) can be expressed as an explicit expression in k but the coefficient function $f_d(k)$ in (19) cannot in general be simplified to an explicit expression in k.

The problem of specifying normal forms or canonical forms for the NTPS representation has not received any attention in the literature. A practical implementation of the NTPS representation has been described by Norman [Nor75] and it is seen to offer some advantages over the TPS representation. However the question of normal forms is left at the TPS level in the sense that the objects ultimately seen by the user are TPS representations, and we have already seen that TPS normal or canonical forms are readily obtained. While a true normal form for the NTPS representation in its most general form is impossible (because such a normal form would imply a solution to the zero equivalence problem for a very general class of expressions), it would be of considerable practical interest to have a canonical form for some reasonable subset of all possible coefficient function specifications. (For example, see Exercises 3-7 and 3-8 for some special forms of coefficient function specifications which can arise).

Extended Power Series

Recall that a field F<x> of extended power series over a coefficient field F can be identified with the quotient field F((x)) of a power series domain F[[x]]. It was shown in section 2.8 that a canonical form for the quotient field F((x)) satisfying conditions (2.75)-(2.77) takes the form

(20) $\frac{a(x)}{x^n}$

where $a(x) \in$ F[[x]] and $n \geqslant 0$. Thus normal and canonical forms for extended power series are

obtained directly from the forms chosen for representation of ordinary power series. The representation of an extended power series can be viewed as the representation of an ordinary power series plus an additional piece of information specifying the value of n in (20).

3.5. DATA STRUCTURES FOR MULTIPRECISION INTEGERS AND RATIONAL NUMBERS

We turn now to the data structure level of abstraction. Before discussing data structures for polynomials in a domain D[x] or rational functions in a field D(x), it is necessary to determine what data structures will be used for the representation of objects in the coefficient domain D. We consider two possible choices for D: the integral domain **Z** of integers and the field **Q** of rational numbers.

Multiprecision Integers

A typical digital computer has hardware facilities for storing and performing arithmetic operations upon a basic data type which is usually called 'integer' and which we shall call *single-precision integer*. The range of values for a single-precision integer is limited by the number of distinct encodings that can be made in the computer *word*, which is typically 8, 16, 32, 36, 48, or 64 bits in length. Thus the value of a signed single-precision integer cannot exceed about 9 or 10 decimal digits in length for the middle-range word sizes listed above or about 19 decimal digits for the largest word size listed above. These restricted representations of objects in the integral domain **Z** are not sufficient for the purposes of symbolic computation.

A more useful representation of integers can be obtained by imposing a data structure on top of the basic data type of 'single-precision integer'. A *multiprecision integer* is a linear list $(d_0, d_1, \ldots, d_{l-1})$ of single-precision integers which represents the value

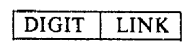
$$\sum_{i=0}^{l-1} d_i \beta^i$$

where the base β has been pre-specified. Noting that single-precision integers can be either positive or negative, a positive multiprecison integer will be represented by a list of positive single-precision integers and a negative multiprecision integer will be represented by a list of negative single-precision integers. (Alternatively, rather than have the sign of the multiprecision integer redundantly stored in every element of the list one might choose some other convention for storing the sign but such differences in detail need not concern us here).

The *base* β could be, in principle, any positive integer greater than 1 such that $\beta-1$ is a single-precision integer, but for efficiency β would be chosen to be a *large* such integer. Two common choices for β are (i) β such that $\beta-1$ is the largest positive single-precision integer (e.g. $\beta = 2^{35}$ if the (signed) word size is 36 bits), and (ii) $\beta = 10^p$ where p is chosen as large as possible such that $\beta-1$ is a single-precision integer (e.g. $\beta = 10^{10}$ if the word size is 36 bits). The *length* l of the linear list used to represent a multiprecision integer may be dynamic (i.e. chosen appropriately for the particular integer being represented) or static (i.e. a pre-specified fixed length), depending on whether the linear list is implemented using linked allocation or using array (sequential) allocation.

Linked Allocation and Array Allocation

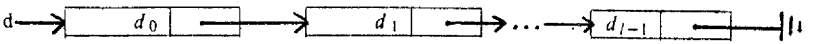
One common method of implementing the linear list data structure for multiprecision integers uses a linked list where each *node* in the linked list is of the form



The DIGIT field contains one base-β digit (a single-precision integer) and the LINK field contains a pointer to the next node in the linked list (or an 'end of list' pointer). Thus the multiprecision integer $d = (d_0, d_1, \ldots, d_{l-1})$ with value

$$(21) \quad d = \sum_{i=0}^{l-1} d_i \beta^i$$

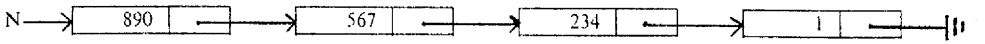
is represented by the linked list



where the 'grounded' notation represents the 'end of list' pointer. Note that the order in which the β-digits d_i are linked is in reverse order compared with the conventional way of writing numbers. For example if $\beta = 10^3$ then the decimal number

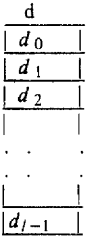
$$(22) \quad N = 1234567890$$

is represented by the linked list



This ordering corresponds to the natural way of writing the base-β expansion of a number as in (21) and, more significantly, it corresponds to the order in which the digits are accessed when performing the operations of addition and multiplication on integers.

Another standard method of implementing a linear list uses array (sequential) allocation. In this scheme the length l of the allowable multiprecision integers is a pre-specified constant and every multiprecision integer is allocated an array of length l (i.e. l sequential words in the computer memory). Thus the multiprecision integer d in (21) is represented by the array



where it should be noted that every multiprecision integer must be expressed using l β-digits (by artificially introducing zeros for the high-order terms in (21) if necessary). For example if $\beta = 10^3$ and $l = 10$ then integers not exceeding 30 decimal digits in length can be represented and the particular decimal number N in (22) is represented by the array

N

890
567
234
1
0
0
0
0
0
0

Advantages and Disadvantages

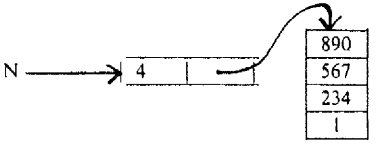
There are some well-known factors affecting the choice between linked and sequential allocation. Sequential allocation requires that the length l be pre-specified and this leads to two significant disadvantages: (i) a decision must be made as to the maximum length of integers that will be required (with a system failure occurring whenever this maximum is exceeded), and (ii) a considerable amount of memory space is wasted storing high-order zero digits (with a corresponding waste in processor time accessing irrelevant zero digits). Linked allocation avoids these problems since irrelevant high-order zero digits are not represented and the length of the list is limited only by the total storage pool available to the system.

On the other hand the use of linked lists also involves at least two disadvantages: (i) a considerable amount of memory space is required for the pointers, and (ii) the processing time required to access successive digits is significantly higher than for array accesses. These two disadvantages would seem to be especially serious for this particular application of linked lists because the need for pointers could potentially double the amount of memory used by multiprecision integers and also because the digits of an integer will *always* be stored and accessed in sequence. However the advantage of *indefinite-precision integers* (i.e. multiprecision integers with dynamically determined length l) along with the even greater advantages of linked allocation for representing polynomials and more general classes of functions makes linked allocation the choice in most systems for symbolic computation. Of the major systems, ALTRAN uses array allocation while SAC-1 and LISP-based systems such as MACSYMA, REDUCE, and SCRATCHPAD use linked allocation.

A third possible implementation for multiprecision integers is the *descriptor allocation* which attempts to combine the best features of the above two implementations. In this scheme one uses a *descriptor block*

l	LINK

which is a node containing the length l of the particular multiprecision integer being represented and a LINK field which contains a pointer to the *array block*, which is an array of the l β-digits. For example if $\beta = 10^3$ then the number N in (22) would be represented by the following scheme:



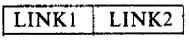
This scheme would seem to avoid all of the disadvantages outlined above for the other two implementations. However there are problems associated with it, including the need for dynamic

array allocation and the need for a sophisticated garbage collection algorithm. While none of the systems uses this scheme for multiprecision integers, the idea can be generalized to a scheme for representing multivariate polynomials which is used by the ALTRAN system.

Rational Numbers

The field **Q** of rational numbers is the quotient field of the integral domain **Z** of integers. A natural representation for rational numbers is therefore the pair *(numerator, denominator)* where each of *numerator* and *denominator* is a multiprecision integer. The basic data structure is a list of length two each element of which is itself a linear list. The representation is made canonical by imposing conditions (2.75) - (2.76) of section 2.7 (condition (2.77) is automatic since the representation for multiprecision integers will be unique).

If multiprecision integers are represented by either linked allocation or array allocation, a rational number can be represented by a node



where LINK1 is a pointer to the numerator multiprecision integer (either a linked list or an array) and similarly LINK2 is a pointer to the denominator. In the case of array allocation for multiprecision integers it is also possible to represent a rational number by a two-dimensional array (e.g. with *l* rows and 2 columns) since the length *l* of the numerator and denominator is a fixed constant.

3.6. DATA STRUCTURES FOR POLYNOMIALS, RATIONAL FUNCTIONS, AND POWER SERIES

Relationships between Form and Data Structure

The data structures used to represent multivariate polynomials in a particular system influences (or conversely, is influenced by) some of the choices made at the form level of abstraction. Referring to the hierarchy illustrated in *Figure 3.1* of section 3.3, the choice made at form level A (normal/canonical forms) is independent of the basic data structure to be used. At form level B the choice between the recursive representation and the distributive representation is in practice closely related to the choice of basic data structure. The recursive representation is the common choice in systems using a linked list data structure while the distributive representation is found in systems using an array (or descriptor block) data structure. (Note however that these particular combinations of choice at form level B and the data structure level are not the only possible combinations). At form level C the sparse representation is the choice in all of the major systems for reasons previously noted and this fact is reflected in the details of the data structure. The choice at form level D regarding the representation of zero exponents is more variable among systems. In systems using the distributive representation it is most natural to explicitly store zero exponents but in systems using the recursive representation the choice is somewhat arbitrary. In any case, the choice is reflected in the details of the data structure.

In this section we describe two possible data structures for multivariate polynomials. The first is a linked list data structure using the recursive, sparse representation and we arbitrarily choose to explicitly represent zero exponents. The second is an array (descriptor block) data structure using the distributive, sparse representation with zero exponents explicitly represented. We describe these two data structures as they apply to multivariate polynomials in expanded canonical form. Then we describe the additional structure which can be imposed on either of these two basic data structures to allow for the implementation of the factored normal form.

A Linked List Data Structure

Using the recursive representation of multivariate polynomials in expanded canonical form, a polynomial domain $D[x_1, \ldots, x_v]$ is viewed as the domain $D[x_2, \ldots, x_v][x_1]$ and this view is applied recursively to the 'coefficient domain' $D[x_2, \ldots, x_v]$. With this point of view, a polynomial $a(x_1, \ldots, x_v) \in D[x_1, \ldots, x_v]$ is considered at the 'highest level' to be a univariate polynomial in x_1 and it can be represented using a linked list where each node in the linked list is of the form

COEF_LINK	EXPONENT	NEXT_LINK

Each such node represents one polynomial term $\cdot a_i x_1^i$ with $a_i \in D[x_2, \ldots, x_v]$, where the EXPONENT field contains the value i (as a single-precision integer), the COEF_LINK field contains a pointer to the coefficient a_i of x_1^i, and the NEXT_LINK field contains a pointer to the next term in the polynomial (or an 'end of list' pointer). This representation is applied recursively. In order to know the name of the indeterminate being distinguished at each level of this recursive representation, we can use a 'header node'

INDET_LINK	FIRST_LINK

where the INDET_LINK field contains a pointer to the name of the indeterminate and the FIRST_LINK field contains a pointer to the first term in the polynomial (at this specific level of recursion).

Example 3.4.

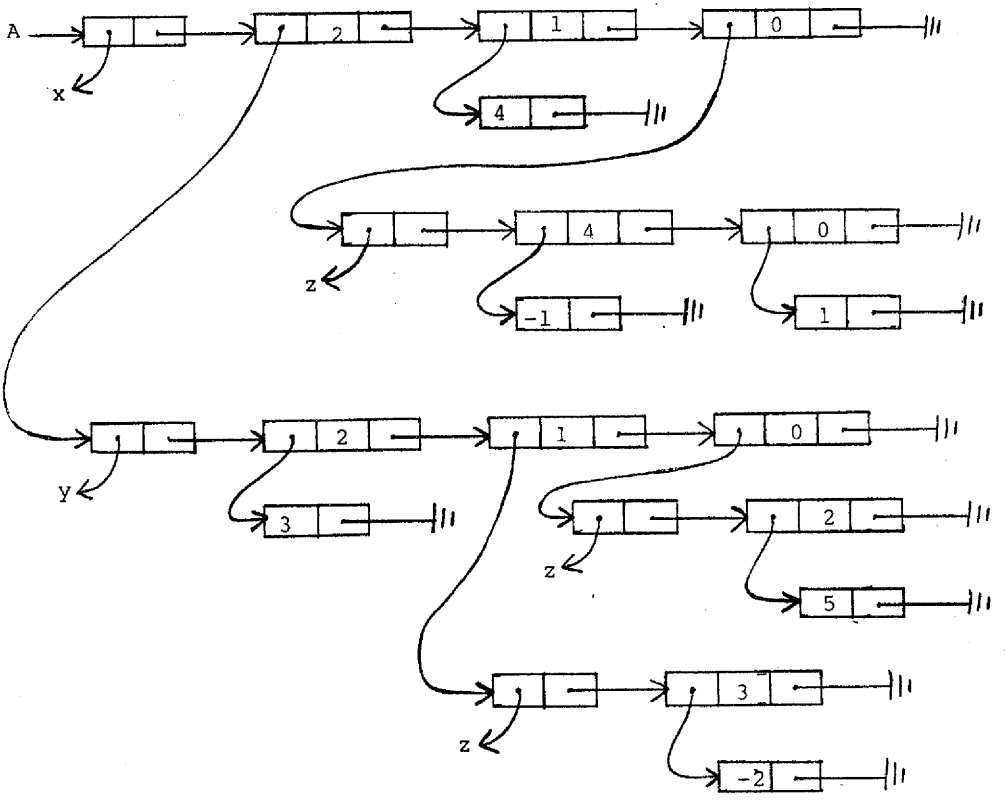
Let $A(x,y,z) \in \mathbf{Z}[x,y,z]$ be the polynomial

$$A(x,y,z) = 3x^2y^2 - 2x^2yz^3 + 5x^2z^2 + 4x - z^4 + 1$$

or, in recursive representation,

$$A(x,y,z) = (3y^2 + (-2z^3)y + 5z^2)x^2 + (4)x + (-z^4 + 1).$$

Using the linked list data structure just described, the polynomial $A(x,y,z)$ is represented as follows.



□

In Example 3.4 the elements in the coefficient domain **Z** are all represented as single-precision integers. Clearly the occurrence of a node representing an integer in this linked list structure could as well be a multiprecision integer in its linked list representation. More generally, the coefficient domain could be the field **Q** of rational numbers in which case rather than an integer node (or list of nodes) there would be a header node for a rational number in its linked list representation, pointing to a pair of multiprecision integers.

In a high-level list processing language using the linked list data structure presented here, it would be possible to distinguish the cases when a pointer is pointing to a polynomial, a multiprecision integer, or a rational number. A polynomial is distinguished by a header node the first field of which points to the name of an indeterminate. A multiprecision integer is distinguished by the fact that the first field of its header node contains a single-precision integer rather than a pointer. A rational number is distinguished by a header node the first field of which points to a multiprecision integer.

A Descriptor Block Data Structure

Among the major systems for symbolic computation the only one which does not use a linked list data structure for representing multivariate polynomials is ALTRAN. The data structure used by ALTRAN is not purely sequential allocation since that would require the dense representation which is not a practical alternative for multivariate polynomials (see section 3.3). ALTRAN uses what can be called a descriptor block data structure which we now describe.

Using the distributive representation of multivariate polynomials in expanded canonical form, a polynomial $a(\mathbf{x}) \in D[\mathbf{x}]$ is viewed in the form

$$a(\mathbf{x}) = \sum_{\mathbf{e} \in \mathbf{N}^v} a_{\mathbf{e}} \mathbf{x}^{\mathbf{e}}$$

where $a_{\mathbf{e}} \in D$, $\mathbf{x} = (x_1, \ldots, x_v)$ is a vector of indeterminates and each $\mathbf{e} = (e_1, \ldots, e_v)$ is a corresponding vector of exponents. More explicitly, a term $a_{\mathbf{e}}\mathbf{x}^{\mathbf{e}}$ is of the form

$$a_{\mathbf{e}} x_1^{e_1} x_2^{e_2} \cdots x_v^{e_v}.$$

With this point of view, the representation of a polynomial $a(\mathbf{x})$ can be accomplished by storing three *blocks* of information: (i) a *layout block* which records the names of the indeterminates $x_1, \ldots, x_v$; (ii) a *coefficient block* which records the list of all nonzero coefficients $a_{\mathbf{e}}$ and (iii) an *exponent block* which records the list of exponents vectors $(e_1, \ldots, e_v)$, one such v-vector corresponding to each coefficient $a_{\mathbf{e}}$ in the coefficient block. The order of the integers in the exponent vectors corresponds to the order of the indeterminates specified in the layout block.

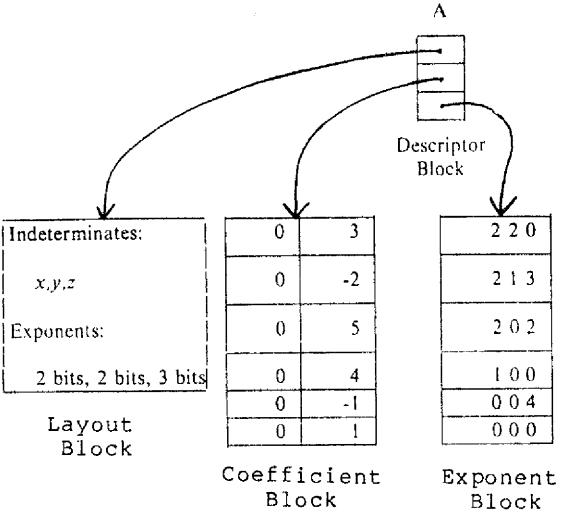
Each block of information is stored as an array (or more specifically in ALTRAN, a block of sequential locations in a large array called the *workspace*). The use of sequential allocation of storage imposes the requirement that the precise 'width' of each block of storage be pre-specified. Thus in the ALTRAN language each variable is associated with a declared layout which specifies the names of all indeterminates which may appear in expressions assigned to the variable and the maximum degree to which each variable may appear. The layout blocks are therefore specified by explicit declarations and the size of each exponent block to be allocated is also known from the declarations. ALTRAN exploits the fact that the maximum size specified for each individual exponent e_i in an exponent vector $\mathbf{e}$ will generally be much smaller than the largest single-precision integer and hence several exponents can be *packed* into one computer word. The layout block is used to store detailed information about this packing of exponents into computer words.

The 'width' of the coefficient block is determined by the range of values allowed for the coefficient domain D. In ALTRAN only multiprecision integers are allowed as coefficients. (Thus the domain **Q[x]** is not represented in ALTRAN but since **Z(x)**, the quotient field of **Z[x]**, will be represented there is no loss in generality). Since ALTRAN uses the array representation of

multiprecision integers (see section 3.5) the length l (in computer words) of all multiprecision integers is a pre-specified constant. The coefficient block therefore consists of l computer words for each coefficient a_e to be represented. Finally, a polynomial a(**x**) is represented by a *descriptor block* which is an array containing three pointers, pointing respectively to the layout block, the coefficient block, and the exponent block for a(**x**).

Example 3.5.

Let $A(x,y,z) \in \mathbf{Z}[x,y,z]$ be the polynomial given in Example 3.4. Using the descriptor block data structure just described, suppose that the declared maximum degrees are degree 2 in x, degree 3 in y, and degree 4 in z. Suppose further that multiprecision integers are represented using base $\beta = 10^3$ and with pre-specified length $l = 2$. Then the polynomial $A(x,y,z)$ is represented as follows.



The layout block illustrated in this example indicates that the information stored in the actual layout block would include pointers to the names of the indeterminates and also a specification of the fact that each vector of three exponents is packed into one computer word, with the exponent of x occupying 2 bits, the exponent of y occupying 2 bits, and the exponent of z occupying 3 bits. In practice there is also a guard bit in front of each exponent (to facilitate performing arithmetic operations on the exponents) so this specification implies that the computer word consists of at least 10 bits. The coefficient block illustrated here reflects the specification of $l = 2$ words for each multiprecision integer although $l = 1$ would have sufficed in this particular example. □

Implementing Factored Normal Form

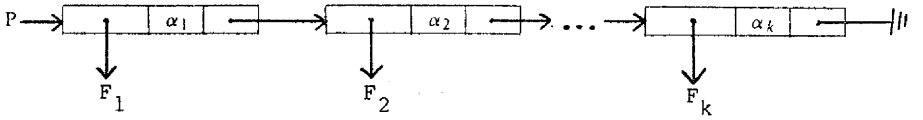
The two basic data structures of this section have been described as they apply to the representation of multivariate polynomials in expanded canonical form. It is not difficult to use either of these basic data structures for the representation of polynomials in a non-canonical normal form or indeed in a non-normal form. The case of the factored normal form will be briefly

examined here.

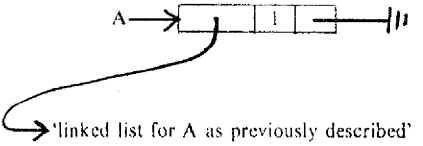
A polynomial P in factored normal form as defined in Definition 3.5 of section 3.3 can be expressed as a product of factors

$$(23) \quad P = \prod_{i=1}^{k} F_i^{\alpha_i}$$

where α_i $(1 \leqslant i \leqslant k)$ is a positive integer, F_i $(1 \leqslant i \leqslant k)$ is a polynomial in expanded canonical form, and $F_i \neq F_j$ for $i \neq j$. Using a linked list data structure, the polynomial P in the product form (23) can be represented by the linked list

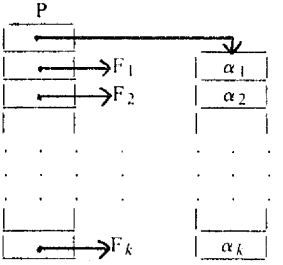


where each factor F_i $(1 \leqslant i \leqslant k)$ is represented by a linked list as previously described for polynomials in expanded canonical form. In a system using this scheme all polynomials are assumed to be in product form and if A is a single polynomial factor in expanded canonical form then it is represented by



Thus we have simply introduced more structure on top of the original linked list data structure for representing multivariate polynomials.

A similar scheme can be used to represent the product form (23) based on a descriptor block data structure. Indeed the ALTRAN system uses the factored normal form as its basic polynomial form. In ALTRAN the polynomial P in the product form (23) is represented by a *formal product block* which is an array containing one pointer to each factor F_i $(1 \leqslant i \leqslant k)$ and an additional pointer to a corresponding array of the powers α_i $(1 \leqslant i \leqslant k)$. Thus the representation of P is



where each factor F_i $(1 \leqslant i \leqslant k)$ is represented by the descriptor block data structure as previously described for polynomials in expanded canonical form.

Rational Functions

A field $D(x_1, \ldots, x_v)$ of rational functions has a natural representation as the quotient field of a polynomial domain $D[x_1, \ldots, x_v]$. In this point of view a rational function is represented by a pair *(numerator, denominator)* where each of *numerator* and *denominator* is a representation of a polynomial lying in the domain $D[x_1, \ldots, x_v]$. Using either a linked list data structure or a descriptor block data structure for polynomials, a rational function is thus represented by a node

LINK1	LINK2

of pointers to the numerator and denominator polynomials. As discussed in section 3.4 a rational function would usually be represented with numerator and denominator relatively prime and with unit normal denominator. In addition, a system would provide one or more (possibly independent) choices of normal forms for the numerator and for the denominator.

A slightly different representation for rational functions is obtained by a trivial generalization of the formal product representation described for polynomials. In the formal product

$$P = \prod_{i=1}^{k} F_i^{\alpha_i}$$

if we allow the powers α_i to be negative integers as well as positive integers then we immediately have a data structure for representing rational functions. No change is required in the data structures already described for formal product representation. This is the data structure used for rational functions in the ALTRAN system. In this formal product representation, the numerator consists of all factors with positive powers and the denominator consists of all factors with negative powers. In the ALTRAN system the denominator is always a unit normal polynomial and various options are available for choosing among the various normal forms discussed in section 3.4.

Power Series

We are considering in this book univariate power series lying in a domain $D[[x]]$ where the coefficient domain D is one of the domains previously discussed (i.e. integers, rational numbers, polynomials, or rational functions). A data structure for power series representation is therefore an extension of data structures previously discussed.

If the TPS representation of power series is used, the TPS

$$\sum_{k=0}^{t} a_k x^k + O(x^{t+1})$$

has a natural representation as a linear list

$$(a_0, a_1, \ldots, a_t).$$

This linear list is easily implemented as either a linked list or an array of pointers to the coefficients a_k in their appropriate representation. As a linked list the 'sparse representation' would be natural (i.e. with only nonzero terms stored) while as an array the 'dense representation' would be used. In the dense representation the truncation degree t is implicitly specified by the fact that there are $t+1$ elements in the linear list, while in the sparse representation the value of t must be stored as an additional piece of information.

The non-truncated representations of power series can be implemented using a similar data structure. The power series

$$a(x) = \sum_{k=0}^{\infty} f_a(k) x^k$$

can be represented as a linear list

$$(a_0, a_1, \ldots, a_{t-1}, f_a(k))$$

where the number l of coefficients which have been explicitly computed may increase as a computation proceeds. Again this linear list can be implemented using either a linked list or an array of pointers, with all but the last element pointing to explicit representations of coefficients and with the last element pointing to a representation of the coefficient function $f_a(k)$. We note that in general the representation of the coefficient function $f_a(k)$ will involve expressions that are much more complicated than we have so far discussed.

Representations for extended power series are obtained by straightforward generalizations of the representations for ordinary power series. As noted at the end of section 3.4, the representation of an extended power series $a(x)$ with coefficients lying in a field F can be viewed as the representation of an ordinary power series plus an additional piece of information specifying the power of x by which the ordinary power series is to be 'divided'. Thus if a particular data structure is chosen for ordinary power series, a data structure for extended power series is obtained by allowing for the representation of one additional (single-precision) integer.

BIBLIOGRAPHY FOR CHAPTER 3

W. S. Brown, On computing with factored rational expressions. Proc. EUROSAM 1974, *ACM SIGSAM Bull.* 8(3), Aug. 1974, pp. 26-34.

B. F. Caviness, On canonical forms and simplification. *J. Assoc. Comput. Mach.* 17(2), Apr. 1970, pp. 385-396.

A. D. Hall, Jr., The Altran system for rational function manipulation -- A survey. *Comm. ACM* 14(8), Aug. 1971, pp. 517-521.

A. C. Hearn, Polynomial and rational function representations. *Technical Report UCP-29*, Univ. of Utah, July 1974.

E. Horowitz and S. Sahni, *Fundamentals of Data Structures.* Computer Science Press, Potomac, Maryland, 1976.

D. E. Knuth, *The Art of Computer Programming, vol. 1: Fundamental Algorithms*, 2nd ed. Addison-Wesley, Reading, Mass., 1975.

J. Moses, Algebraic simplification: A guide for the perplexed. *Comm. ACM* 14(8), Aug. 1971, pp. 527-537.

A. C. Norman, Computing with formal power series. *ACM Trans. Math. Software* 1(4), Dec. 1975, pp. 346-356.

EXERCISES

3-1. For each of the following expressions try to find the simplest equivalent expression. As a measure of 'simplicity' one could count the number of characters used to write the expression but a higher level measure such as the number of 'terms' in the expression would suffice.

(a) $a(x,y) = ((x^2 - xy + x) + (x^2 + 3)(x - y + 1)) ((y^3 - 3y^2 - 9y - 5) + x^4(y^2 + 2y + 1))$.

(b) $b(x,y) = (x - y)(x^9 + x^8y + x^7y^2 + x^6y^3 + x^5y^4 + x^4y^5 + x^3y^6 + x^2y^7 + xy^8 + y^9)$.

(c) $c(x,y) = \dfrac{(x - y)}{b(x,y)}$, where $b(x,y)$ is the polynomial defined in part (b).

(d) $$d(x) = \frac{e^x \cos x + \cos x \sin^4 x + 2\cos^3 x \sin^2 x + \cos^5 x}{x^2 - x^2 e^{-2x}} - \frac{e^{-x}\cos x + \cos x \sin^2 x + \cos^3 x}{x^2 e^x - x^2 e^{-x}}.$$

3-2. Determine whether or not each of the following expressions is equivalent to zero.

(a) $$a(x,y) = \frac{2x}{3y^3(x^2 - y^2)} + \frac{x - y}{x^5 + x^4y + x^3y^2 + x^2y^3 + xy^4 + y^5} - \frac{x + y}{3y^3(x^2 - xy + y^2)} - \frac{x - y}{3y^3(x^2 + xy + y^2)} - \frac{x^2 + y^2}{x^6 - y^6}.$$

(b) $$b(x,y) = \frac{x - y}{x^5 + x^4y + x^3y^2 + x^2y^3 + xy^4 + y^5} - \frac{x^2 - xy + y^2}{x^6 - y^6}.$$

(c) $$c(x) = 16 \sin^{-1}(x) + 2 \cos^{-1}(2x) - 3 \sinh^{-1}(\tan(\tfrac{1}{2}x)).$$

(d) $$d(x) = 16 \cos^3(x) \cosh(\tfrac{1}{2}x) \sinh(x) - 6 \cos(x) \sinh(\tfrac{1}{2}x) - 6 \cos(x) \sinh(\tfrac{3}{2}x) - \cos(3x)(e^{\frac{3}{2}x} + e^{\frac{1}{2}x})(1 - e^{-2x}).$$

3-3. In this problem you will show that, in a certain sense, if the zero equivalence problem can be solved for a given class of expressions then the general simplification problem can also be solved. Let E be a class of expressions and let f be a normal function defined on E. Suppose there is an algorithm A which will generate all of the syntactically valid expressions in the class E, in lexicographically increasing order. (i.e. Algorithm A generates all syntactically valid expressions containing l characters, for $l = 1$, then $l = 2$, then $l = 3$, etc. and the expressions of a fixed length l are generated in increasing order with respect to some encoding of the characters).

(a) Define a *simplification function* g on E in terms of normal function f and algorithm A such that g is a canonical function and moreover the canonical form $g(a)$ of any expression $a \in E$ is the *shortest* expression equivalent to a.

(b) If E is a class of expressions obtained by performing the operations of addition and multiplication on the elements in a quotient field Q(D) of an integral domain D then the usual canonical function (i.e. 'form a common denominator' and 'reduce to lowest terms') is not a simplification in the sense of the function g of part (a). Illustrate this fact for the field **Q** of rational numbers by giving examples of expressions of the form $\frac{a_1}{b_1} + \frac{a_2}{b_2} + \frac{a_3}{b_3}$ (where $a_i, b_i \in \mathbf{Z}$ and $\frac{a_i}{b_i}$ is in lowest terms, for $i =$ 1,2,3) such that the 'reduced form' of the expression requires more characters for its

representation than the original expression.

3-4. Consider the four forms for rational functions discussed in section 3.4: *factored/factored, factored/expanded, expanded/factored, expanded/expanded,* with numerator and denominator relatively prime in each case.

(a) Put each of the expressions $a(x,y)$, $b(x,y)$, and $c(x,y)$ given in problem 3-1 into each of the above four forms. Similarly for the expressions $a(x,y)$ and $b(x,y)$ given in problem 3-2.

(b) Which (if any) of these four forms is useful for performing 'simplification' as requested in problem 3-1? for determining 'zero-equivalence' as requested in problem 3-2?

3-5. Consider the problem of computing the functions $f(x,y)$ and $g(x,y)$ defined by

$$f(x,y) = \frac{\partial a}{\partial x}\,\frac{\partial b}{\partial x},$$

$$g(x,y) = \frac{\partial a}{\partial y}\,\frac{\partial b}{\partial y}$$

(where ∂ denotes partial differentiation) where a and b are the rational functions

$$a = \frac{(10x^2y^3 + 13x - 7)(3x^2 - 7y^2)^2}{(5x^2y^2 + 1)^2 (x - y)^3 (x + y)^2},$$

$$b = \frac{13x^3y^3 + 75x^3y + 81xy - x + 19}{(5x^2y^2+1)(x - y)(x + y)^3}.$$

Perform this computation on a system (or systems) available to you using several different choices of normal (or non-normal) forms available in the system. Compare the results obtained using the various choices of form in terms of (i) processor time used, (ii) memory space required, and (iii) compactness (i.e. readability) of the output.

3-6. The TPS representation of a power series appears to be similar to a polynomial but must be distinguished from a polynomial. Consider the two power series defined by

$$a(x) = \sum_{k=0}^{\infty} (-1)^k x^k \qquad \left[= \frac{1}{1+x}\right],$$

$$b(x) = \sum_{k=0}^{\infty} \frac{1}{k!}\, x^k \qquad \left[= e^x\right].$$

(a) The TPS representations of $a(x)$ and $b(x)$ with truncation degree $t = 3$ are

$$\tilde{a}(x) = 1 - x + x^2 - x^3 + O(x^4),$$

$$\tilde{b}(x) = 1 + x + \frac{1}{2}x^2 + \frac{1}{6}x^3 + O(x^4).$$

Let $p(x)$ and $q(x)$ be the corresponding polynomials defined by

$$p(x) = 1 - x + x^2 - x^3,$$

$$q(x) = 1 + x + \frac{1}{2}x^2 + \frac{1}{6}x^3.$$

What should be the result of performing the TPS multiplication $\tilde{a}(x)\tilde{b}(x)$? What is

the result of performing the polynomial multiplication $p(x)\ q(x)$? What is the correct power series product $a(x)\ b(x)$ expressed as a TPS with truncation degree $t = 6$?

(b) Let $\tilde{a}(x)$ and $p(x)$ be as in part (a). The result of performing $\frac{1}{p(x)}$ is the rational function $\frac{1}{(1 - x + x^2 - x^3)}$. What is the result of performing the TPS division $\frac{1}{\tilde{a}(x)}$? What is the correct power series reciprocal $\frac{1}{a(x)}$?

3-7. Consider the problem of computing the power series solution of a linear ordinary differential equation with polynomial coefficients:

$$p_\nu(x)y^{(\nu)} + \cdots + p_1(x)y' + p_0(x)y = r(x)$$

where $p_i(x)$, $0 \leqslant i \leqslant \nu$, and $r(x)$ are polynomials in x and where y denotes the unknown function of x. Show that if this differential equation has a power series solution

$$y(x) = \sum_{k=0}^{\infty} y_k x^k$$

then the power series coefficients can be expressed, for $k \geqslant K$ for some K, as a finite linear recurrence:

$$y_k = u_1(k)y_{k-1} + u_2(k)y_{k-2} + \cdots + u_n(k)y_{k-n}$$

where $u_i(k)$, $1 \leqslant i \leqslant n$, are rational expressions in k. Thus an NTPS representation for the solution $y(x)$ is possible with the coefficient function $f_y(k)$ specified by a finite linear recurrence (and with the first k coefficients specified explicitly).

3-8. Show that a power series $a(x) = \sum_{k=0}^{\infty} a_k x^k$ has an NTPS representation in which the coefficient function can be expressed, for $k \geqslant K$ for some K, as a finite linear recurrence with *constant coefficients:*

$$a_k = u_1 a_{k-1} + u_2 a_{k-2} + \cdots + u_n a_{k-n}$$

if and only if $a(x)$ can be expressed as a rational function of x. (See section 2.8).

3-9. Generalize the results of problem 3-7 and of problem 3-8 into statements about the NTPS representation of extended power series rather than just ordinary power series.

3-10. Using a language in which linked list manipulation is convenient, implement algorithms for addition and multiplication of indefinite-precision integers (i.e. multiprecison integers in linked list representation). Base your algorithms on the methods you use to do integer arithmetic by hand.

3-11. Assuming that the algorithms of problem 3-10 are available, implement algorithms for addition and multiplication of multivariate polynomials in expanded canonical form with indefinite-precision integer coefficients. Use the linked list data structure described in section 3.6. Base your algorithms on the methods you use to do polynomial arithmetic by hand.

3-12. Assuming that the algorithms of problems 3-10 and 3-11 are available, implement algorithms for addition and multiplication of multivariate rational functions in *expanded/expanded* form (i.e. in the expanded canonical form of Definition 3.6) with indefinite-precision integer coefficients. Use either of the linked list data structures for rational functions described in section 3.6. You will need a recursive implementation of Algorithm 2.3 (or some other algorithm for GCD computation).

3-13. Assuming that the algorithms of problem 3-12 are available, implement algorithms for addition and multiplication of univariate power series with coefficients which are multivariate rational functions. Use the TPS representation implemented as a linked list.

3-14. Choose a specific representation for extended power series and implement algorithms for addition, multiplication, and division of extended power series with coefficients which are multivariate rational functions. You will need to have available the algorithms of problem 3-12 and you may wish to have available the algorithms of problem 3-13.