Polynomial GCD using straight line program representation

Naylor, W. A.

Award date:
2000

Awarding institution:
University of Bath

POLYNOMIAL GCD USING STRAIGHT LINE PROGRAM REPRESENTATION

Submitted by Bill Naylor
for the degree of
Doctor of Philosophy
of the University of Bath
2000

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Contents

0.1 Notation List

SLI	Straight-line Instruction see section 3.2
SLP	Straight-line Program see section 3.3
SLV	Straight-line Value see section 3.13
$K[a,b]$	traditional polynomials represented in sparse form.
$K(a,b)$	traditional rational functions.
$K(a,b)\{x,y\}$	SLPs in x,y over $K(a,b)$
$K\{x,y\}(a,b)$	sparse form polynomials in a,b with SLP coefficients in x,y.
$deg_x(p)$	The degree of the polynomial p, in the variable x
$p_{proj}(x_1,\cdots,x_i)$	The projection of p to $\mathbf{R}^i$
$\mathbf{0}_n$	$\underbrace{0,\cdots,0}_{n}$
cont(p)	The content of p
pp(p)	The primitive part of p
WLOG	Without Loss Of Generality
s.t.	Such That
iff	if and only if
$f \circ g$	the function f composed with the function g

0.2 Summary

This thesis is concerned with calculating polynomial greatest common divisors using straight line program representation.
In the Introduction chapter, we introduce the problem and describe some of the traditional representations for polynomials, we then talk about some of the general subjects central to the thesis, terminating with a synopsis of the category theory which is central to the AXIOM computer algebra system used during this research.
The second chapter is devoted to describing category theory. We follow with a chapter detailing the important sections of computer code written in order to investigate the straight line program subject. The following chapter on evaluation strategies and algorithms which are dependant on these follows, the major algorithm which is dependant on evaluation and which is central to our thesis being that of equality checking. This is indeed central to many mathematical problems. Interpolation, that is the determination of coefficients of a polynomial is the subject of the next chapter. This is very important for many straight line program algorithms, as their non-canonical structure implies that it is relatively difficult to determine coefficients, these being the basic objects that many algorithms work on. We talk about three separate interpolation techniques and compare their advantages and disadvantages. The final two chapters describe some of the results we have obtained from this research and finally conclusions we have drawn as to the viability of the straight line program approach and possible extensions.
Finally we terminate with a number of appendices discussing side subjects encountered during the thesis.

0.3 Acknowledgements

Firstly I would like to acknowledge and thank my supervisor Professor James Davenport for his help, encouragement and support during my research. I thank Numerical Algorithms Group Ltd. (NAG) firstly for their financial support, and secondly for providing the AXIOM system which was used extensively in fact exclusively as the system in which the computational part of the research was performed. I would specifically like to thank Mike Dewar, Peter Broadbery and Themos Tsikas, members of the NAG Computational Mathematics Group for their advise on how to circumvent the occasional quirkyness of AXIOM and odd mathematical difficulties. I have Dr. D. Richardson to thank for many discussions about an area which transpired to be one of the major difficult areas of my method, that of equality checking of SLPs. Amongst my contemporaries I would specifically like to thank Jetender Kang (Jet) and Dave Power (Power Dave) for conversations suggesting improvements and alternate algorithms which I might utilise, often over a pint of ale or two. I have many other friends I would like to thank who helped me through the trials and tribulations of student life. Last but not least I would like to thank the Parade Bar for providing a fine pint of BarnStormer (most of the time), which constituted a fitting end to a hard days graft!

Chapter 1

Introduction

1.1 Introduction to the problem

We may use Straight-Line Programs [11, 4, 19, 27, 26, 17, 25, 22, 7] as an alternative means of representing polynomials. In this thesis the problem that we consider is specifically the calculation of the greatest common divisor of two polynomials using Straight-Line Program representation. We shall compare and contrast the times taken for performing these calculations and the sizes of the objects formed with the equivalent measurements using traditional representations. We shall also consider how the GCD domain which we shall create fits into the AXIOM category system. We shall consider how good the performance of this domain is when we use it as a parameter domain for calculating Gröbner bases.

1.2 Traditional Polynomial Representations

Polynomials may be represented in various different ways. The form of the representation for polynomials will have a profound implication on the types of algorithm that may be performed on them. In the following we describe some of the traditional representations which are used to represent polynomials, followed by a description of the Straight-Line Program representation which we have implemented.

1.2.1 Dense Representation

We may represent a univariate polynomial in dense representation by storing the coefficients in a one dimensional array with elements from the base field of the polynomial. If x is the variable in the polynomial, the coefficient of x^i is held in the $i+1^{\text{th}}$ element of the array. To access a coefficient requires constant time, so access time is $O(1)$. To store a univariate polynomial will require an array of size $d+1$, where d is the degree of the polynomial.

To represent a multivariate polynomial which has n variables, we require some ordering on the variables in the polynomial. We shall denote these variables $x_1 > x_2 > \cdots > x_n$. Now we may view the coefficients of variable x_1 as polynomials in $\{x_2, \cdots, x_n\}$. In each of these polynomials we view the coefficients of x_2 as polynomials in $\{x_3, \cdots, x_n\}$ and so on until we get to the base field coefficients. With this recursive view in mind we may store a polynomial which has n variables in an n dimensional array. The coefficient of $x_1^j x_2^k \cdots x_n^l$ being stored in position $(j+1, k+1, \cdots, l+1)$ of the array. The access time is again constant ($O(1)$) though the hidden constant will be larger as the position in memory must be determined from a mapping of $(j, k, \cdots, l)$ to memory (an assumption we make here is that multiplication and summation of integers is a constant time operation, this will be true so long as $j, k, \cdots, l$ do not get large, that is so long as $j, k, \cdots, l < 2^W$ where W is the word size for the computer). The size of the array will be $\prod_{i=1}^{n}(d_i+1)$ where d_i is the degree of x_i in the polynomial.

We see that this representation allows very fast retrieval of coefficients, but it does mean that every coefficient including every zero coefficient must be recorded. If there are a lot of zero coefficients this constitutes a large waste of memory.

1.2.2 Sparse Non-recursive Representation

Using sparse representation for a polynomial only the non-zero coefficients are recorded. To this end we may store a polynomial as a List of Records, where the Records have the following structure:

$$\text{Record}(\text{coefficient} : \text{R}, \text{powerProd} : \text{List}(\text{Record}(\text{variable} : \text{Symbol}, \text{power} : \text{I}))) \tag{1.1}$$

In the previous R is the domain of the coefficients, generally an algebraic structure known as a *Ring*, I is the domain of the indices, generally an algebraic structure known as an *Abelian Monoid*, a common example is the Non-Negative Integers. In order to access a coefficient of this polynomial the list must be traversed until the specific element has been found. This operation will have complexity $O(l)$ where l is the number of non-zero coefficients, worst time complexity is $O(l)$, whereas average time is $O(l/2)$. It is possible to cut this time to $O(\log l)$ by imposing an ordering on the records 1.1, then using a 'binary chop' algorithm, for example the well known *quicksort* algorithm. The ordering we use would be an ordering on power products, for example the *lexicographic ordering* [9]. This would be applied to the powerProd part of the records in 1.1. The storage space is now $O(l)$. We note that in many cases l may be far smaller than the possible number of coefficients, viz. $\prod_{i=1}^{n}(d_i + 1)$, as noted earlier.

1.2.3 Sparse Recursive Polynomial Representation

Another good representation for sparse polynomials is the *sparse recursive* representation, where polynomials are represented as a list of records where the records have the following recursive structure:

$$\mathrm{R}_{x_n} = \mathrm{Record}(\mathrm{coefficient} : \mathrm{R}_{x_{n-1}}, \mathrm{variable} : \mathrm{Symbol}, \mathrm{power} : \mathrm{I}) \ , \ \text{for n} \geq 1 \tag{1.2}$$

$$\mathrm{R}_{x_0} = \mathrm{R} \tag{1.3}$$

where I and R have the same meaning as in the previous section. The access time for the coefficient of a polynomial represented in this representation has $O(ln)$, l being the average length of each record, and n being the number of variables. The average time is $O(\frac{ln}{2})$.

1.2.4 Comparison

We see that for dense representation, the accessing of coefficients is very fast, however every coefficient must be stored. Now if an algorithm deals with polynomials stored using dense representation then every coefficient must be considered,

including every zero coefficient. This leads to some large lower bounds to the complexity for many algorithms. For sparse representation, we may have a different problem, this is that even though we have removed the problem of storing and considering zero elements, it is not simple to access coefficients and again gives large time complexities for many algorithms. Sparse representation corresponds to the traditional mathematical notation (without brackets), for example:

$$x^5 + 4x^3 + 7$$

a dense version of this example would be represented as:

$$x^5 + 0x^4 + 4x^3 + 0x^2 + 0x + 7$$

We should note that mathematicians often do use brackets (amongst other expressions) in order to write polynomials, for example:

$$(x^2 + x + 1)(y^2 + y + 1)(z^2 + z + 1) \tag{1.4}$$

which in sparse (or dense) representation has 27 terms. Indeed, as we increase the number of variables, the length of 1.4 is linear in the number of variables but the number of terms in dense or sparse representation is exponential in the number of variables. We look at a third type of representation which has been proposed. It is claimed that this representation has some complexity advantages over the classical dense and sparse representations, this representation uses *Straight-Line Progams* (SLPs) to represent polynomials. We intend to make a practical Straight-Line Progam implementation using the AXIOM computer algebra system, this implementation will perform one of the fundamental operations which is near the centre of any polynomial based computer algebra system, that of finding the *greatest common divisor* (gcd) of two polynomials. We shall then make some benchmark tests to see how the performance compares to the sparse implementation already existent in AXIOM.

1.3 Straight-Line Programs

SLPs may be thought of as directed-acyclic graphs (DAGs), which encode the arithmetic network equivalent to the polynomial.

These DAGs have four types of internal node, *plus*, *minus*, *times* and *quotient* which represent the four basic arithmetic operations, we shall term these *operation nodes*. These DAGs also have two types of leaf nodes, *constant nodes* and *input nodes*, the former represent simple constants from a base field, whilst the latter represent the polynomial variables. An important subset of these DAGs which from now on we shall term SLPs contain no quotient operations, these shall be termed *division free* SLPs. Every node of a given SLP can be thought of as representing a polynomial. If we intend to represent specific polynomials, we must indicate which nodes of the SLP are the ones which represent these polynomials. In our rendering of the SLP we shall indicate this by placing a ">" to the right of the particular node. These special nodes are known as *return nodes.*

Example 1

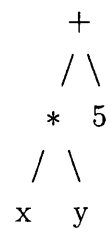
The polynomial $p(x, y)$ represented in sparse form:

$$xy + 5$$

p represented as an SLP :

```
line 1 :  Constant node  : 5
line 2 :  Input node     : x
line 3 :  Input node     : y
line 4 :  Operation node : (times line 2, line 3)
line 5 :  Operation node : (plus line 4, line 1)   >
```

the DAG is :



One thing which must be noticed about SLPs is that in the case of division free SLPs, the representation only covers polynomials where the exponent set is the Non-Negative-Integers. In the case of non-division free SLPs, we may also represent rational functions, extending the domain somewhat. More exotic

expressions, for example logarithmic or exponential expressions, would require some extra basic operations, perhaps an integration operator or an exponentiation operator.

The question which must be posed is "why use SLPs for representing polynomials rather than the more familiar sparse form?". SLPs provide an efficient form of representation for polynomials, especially those which contain many variables to high degrees. For a polynomial of degree deg_i in each of n variables the number of terms which it may have is up to $\prod(deg_i + 1)$ terms. There are operations where it is necessary to consider every one of these terms to deduce a valid answer. We shall look at some examples in order to display some of the advantages that SLPs have over other forms of representations.

Example 2

Consider the multiplication of two randomly generated polynomials, with n_1 and n_2 terms respectively, the result will have size $\approx n_1 n_2$ terms. Now for the same problem where the polynomials are encoded as SLPs of length l_1 and l_2 respectively, the result will have length $\approx l_1 + l_2 + 1$.

Example 3

Storage of the polynomial $(x+1)^{100}$ in dense form requires the storage of 101 coefficients, the largest of which is 100891344545564193334812497256. If we where however to represent the polynomial as an SLP, we could use a program which had one addition node, and eight multiplication nodes viz:

line 1 : Input node : x
line 2 : Constant node : 1
line 3 : Operation node : (plus line 1, line 2)
line 4 : Operation node : (times line 3, line 3)
line 5 : Operation node : (times line 4, line 4)
line 6 : Operation node : (times line 5, line 5)
line 7 : Operation node : (times line 6, line 6)
line 8 : Operation node : (times line 7, line 7)
line 9 : Operation node : (times line 8, line 8)
line 10 : Operation node : (times line 9, line 8)
line 11 : Operation node : (times line 10, line 5) >

Example 4

Now we look at an example of a non-division free SLP:

$$\frac{x^{101}-1}{x-1} = \sum_{i=0}^{100} x^i$$

The sparse form of this polynomial would have 101 terms, we could represent this as an SLP with divisions as follows:

line 1 : InputNode : x
line 2 : OperationNode : (times , line 1 , line 1)
line 3 : OperationNode : (times , line 2 , line 2)
line 4 : OperationNode : (times , line 3 , line 3)
line 5 : OperationNode : (times , line 4 , line 4)
line 6 : OperationNode : (times , line 5 , line 5)
line 7 : OperationNode : (times , line 6 , line 6)
line 8 : OperationNode : (times , line 6 , line 7)
line 9 : OperationNode : (times , line 3 , line 8)
line 10 : OperationNode : (times , line 1 , line 9)
line 11 : ConstantNode : - 1
line 12 : OperationNode : (plus , line 11 , line 10)
line 13 : OperationNode : (plus , line 1 , line 11)
line 14 : OperationNode : (quotient , line 12 , line 13) >

It should be noted that though this is not a division free SLP it could be converted into one. Removing divisions from an SLP is known as *division removal*: we shall discuss a technique of division removal later in section 5.7.

Example 5

This next example uses SLP representation for the determinant of a symbolic matrix of dimension n. We note that we may represent this determinant in $n!$ monomials using sparse representation; using dense representation we could represent the determinant in 2^{n^2} monomials; whereas using SLP representation, we could use as few as n^3 elements, using a method due to Berkowitz [2], or one due to Bareiss [1].

Specific Example

Consider the following matrix:

$$\begin{pmatrix} x_{00} & x_{10} & x_{20} & x_{30} \\ x_{01} & x_{11} & x_{21} & x_{31} \\ x_{02} & x_{12} & x_{22} & x_{32} \\ x_{03} & x_{13} & x_{23} & x_{33} \end{pmatrix}$$

In sparse form the determinant of this matrix may be calculated as:

$$x_{00}x_{11}x_{22}x_{33} - x_{01}x_{10}x_{22}x_{33} - x_{00}x_{12}x_{21}x_{33} + x_{02}x_{10}x_{21}x_{33} + x_{01}x_{12}x_{20}x_{33} - x_{02}x_{11}x_{20}x_{33} - x_{00}x_{11}x_{23}x_{32} + x_{01}x_{10}x_{23}x_{32} + x_{00}x_{13}x_{21}x_{32} - x_{03}x_{10}x_{21}x_{32} - x_{01}x_{13}x_{20}x_{32} + x_{03}x_{11}x_{20}x_{32} + x_{00}x_{12}x_{23}x_{31} - x_{02}x_{10}x_{23}x_{31} - x_{00}x_{13}x_{22}x_{31} + x_{03}x_{10}x_{22}x_{31} + x_{02}x_{13}x_{20}x_{31} - x_{03}x_{12}x_{20}x_{31} - x_{01}x_{12}x_{23}x_{30} + x_{02}x_{11}x_{23}x_{30} + x_{01}x_{13}x_{22}x_{30} - x_{03}x_{11}x_{22}x_{30} - x_{02}x_{13}x_{21}x_{30} + x_{03}x_{12}x_{21}x_{30}$$

This is a sum of 24 monomials each of which is a product of four quantities in total 96 elements. The evaluation of which would require 72 operations.

If we used Bareiss's method [Appendix D], we would be able to represent this determinant by a program which had 37 multiplication nodes, 16 subtraction nodes and five division nodes, in total 52 operations. viz.

line 1 : ConstantNode : 1
line 2 : InputNode : x00
line 3 : InputNode : x10
line 4 : InputNode : x20
line 5 : InputNode : x30
line 6 : InputNode : x01
line 7 : InputNode : x11
line 8 : InputNode : x21
line 9 : InputNode : x31
line 10 : InputNode : x02
line 11 : InputNode : x12
line 12 : InputNode : x22
line 13 : InputNode : x32
line 14 : InputNode : x03
line 15 : InputNode : x13
line 16 : InputNode : x23
line 17 : InputNode : x33
line 18 : OperationNode: (times , line 2, line 13)
line 19 : OperationNode: (times , line 10 , line 5)
line 20 : OperationNode: (minus , line 18 , line 19)
line 21 : OperationNode: (times , line 2, line 16)
line 22 : OperationNode: (times , line 14 , line 4)
line 23 : OperationNode: (minus , line 21 , line 22)
line 24 : OperationNode: (times , line 2, line 9)
line 25 : OperationNode: (times , line 6 , line 5)
line 26 : OperationNode: (minus , line 24 , line 25)
line 27 : OperationNode: (times , line 2, line 15)
line 28 : OperationNode: (times , line 14 , line 3)
line 29 : OperationNode: (minus , line 27 , line 28)
line 30 : OperationNode: (times , line 2, line 17)
line 31 : OperationNode: (times , line 14 , line 5)
line 32 : OperationNode: (minus , line 30 , line 31)
line 33 : OperationNode: (times , line 2, line 8)
line 34 : OperationNode: (times , line 6 , line 4)
line 35 : OperationNode: (minus , line 33 , line 32)
line 36 : OperationNode: (times , line 2, line 11)
line 37 : OperationNode: (times , line 10 , line 3)

```
line 38 : OperationNode   : (minus , line 36 , line 37)
line 39 : OperationNode   : (times , line 2, line 10)
line 40 : OperationNode   : (times , line 10 , line 4)
line 41 : OperationNode   : (minus , line 39 , line 40)
line 42 : OperationNode   : (times , line 2, line 7)
line 43 : OperationNode   : (times , line 6 , line 3)
line 44 : OperationNode   : (minus , line 42 , line 43)
line 45 : OperationNode   : (quotient , line 1 , line 2)
line 46 : OperationNode   : (times , line 44 , line 20)
line 47 : OperationNode   : (times , line 38 , line 26)
line 48 : OperationNode   : (minus , line 46 , line 47)
line 49 : OperationNode   : (times , line 45 , line 48)
line 50 : OperationNode   : (times , line 44 , line 23)
line 51 : OperationNode   : (times , line 29 , line 35)
line 52 : OperationNode   : (minus , line 50 , line 51)
line 53 : OperationNode   : (times , line 45 , line 52 )
line 54 : OperationNode   : (times , line 44 , line 32)
line 55 : OperationNode   : (times , line 29 , line 26)
line 56 : OperationNode   : (minus , line 54 , line 55)
line 57 : OperationNode   : (times , line 45 , line 56)
line 58 : OperationNode   : (times , line 44 , line 41)
line 59 : OperationNode   : (times , line 38 , line 35)
line 60 : OperationNode   : (minus , line 58 , line 59)
line 61 : OperationNode   : (times , line 45 , line 60)
line 62 : OperationNode   : (times , line 2 , line 7)
line 63 : OperationNode   : (times , line 6 , line 3)
line 64 : OperationNode   : (minus , line 62 , line 63)
line 65 : OperationNode   : (times , line 61 , line 57)
line 66 : OperationNode   : (times , line 53 , line 49)
line 67 : OperationNode   : (minus , line 65 , line 63)
line 68 : OperationNode   : (quotient , line 1 , line 64)
line 69 : OperationNode   : (times , line 67 , line 68)   >
```

We note that this does look more verbose than the sparse form, however this is only the way it is printed, not the underlying structure. The improvement would

increase as the size of the matrix became larger.

1.4 Metrics for SLPs

We consider two of the fundamental metrics which are used to measure the 'size' of an SLP, these are:

- *length*: The length of an SLP is defined as the number of elements (the number of nodes in the defining DAG) in the SLP.
- *depth*: The depth of an SLP is defined as the longest contiguous path between the return node and any leaf node in the SLP.

These metrics are important since the depth of an SLP gives the asymptotic time that a parallel computer, with an unbounded number of processors would take to evaluate the SLP. The length of the SLP is a measure of the asymptotic time taken by a computer to perform its sequential evaluation.

1.5 Polynomial Greatest Common Divisor

The problem that we are going to approach is the calculation of a polynomial greatest common divisor (Polynomial GCD), using SLP representation. The technique we shall use is based on that used by Kaltofen [11]. Kaltofen uses a *Monte Carlo* technique (that is a probabilistic technique which returns an answer which is probably correct, in a short time). We initially decided to use a *Las Vegas* technique (that is a technique which returns an answer that is always correct, probably in a short time), the reason being that our function must always be made available to functions which purport to return a deterministic answer and therefore should only use deterministic subfunctions. However, we eventually concluded that deterministic equality checking, a procedure that was essential to our algorithm, was such a difficult problem that we would have to investigate how we could bridge the problem. This is investigated in sections 6.5 and 8.4.

The approach that is taken by many is to make the probability of an incorrect result miniscule.

1.6 The AXIOM computer algebra system and its category system

The computer algebra system which we shall be using for this research is the AXIOM computer algebra system. It has a revolutionary type system which we discuss in more detail in the next chapter. The types themselves have types which are called *Categories.* The categories are arranged in a hierarchical structure. They mimic as closely as possible the mathematical structure which they intend to model. We describe these in the next chapter.

Example 5

The type which we intend to form is called: **GcdMulti(R)** where **R** is an arbitrary Euclidean Domain, though in our implementation we are forced by time restrictions to restrict R to be either IntegerNumberSystem (a model for the Integers) or QuotientFieldCategory (the category of fractions of an integral domain). The exported functions of GcdMulti which characterize a GcdDomain are:

gcd : (%,%) $->$ %
exquo : (%,%) $->$ Union(value1:%,failed:'failed')

In the above list, the syntax is as follows, '%' refers to the domain that the functions are exported from, the name on the left hand side of the ':' is the name of the function, the parenthesis on the left of '$->$' contain the types of the arguments of the function and the type on the right is the type of the value returned by the function.

The function 'gcd' finds the *greatest common divisor* of two objects.
The function 'exquo' returns the *exact quotient* of two objects. By exact quotient, we mean the value such that the following holds:

$$qd = n \wedge \text{exquo}(n, d) = q$$

If such a value does not exist, then the value **'failed'** is returned.

We may now say that GcdMulti is in the category **GcdDomain**, a structure already existent in the AXIOM system. We shall see that there are domains already existing in the AXIOM system which take domain parameters, where the domain must be of a certain category. The functions exported by the parameter domain may then be used in the parameterised domain.

Example 6

The domain **GroebnerPackage** takes as parameters:

(Dom: GcdDomain, Expon: OrderedAbelianMonoidSup, VarSet: OrderedSet, Dpol: PolynomialCategory(Dom,Expon,VarSet))

Expon is the exponent domain, VarSet the variable domain, and Dpol specifies the ordering of the variables. Now Dom is the domain that we are interested in, GroebnerPackage contains functions which calculate the Gröbner basis of a set of polynomials, which define a polynomial ideal. The algorithm used is a form of the Buchberger algorithm, this makes heavy use of the functions gcd and exquo of polynomial coefficients, these functions will be taken from the domain Dom, so in fact we may use the functions we have defined in GcdMulti by specifying for example:

GroebnerPackage(GcdMulti(Integer), Expon, OrderedVariableList [x,y,z], Dmp)

where :

- Expon is DirectProduct(3,NonNegativeInteger)
- DMP is DistributedMultivariatePolynomial([x,y,z],GcdMulti(Integer))

Chapter 2

Domains and Categories

Much of the following chapter we take from Jenks [8], also from Doye [21]. We begin by defining what we mean by the terms *Item*, *Domain* and *Category*, also the concept of a *Functor*.

2.1 Items

These are *first order* objects, they are elements of Domains see section 2.2.
Examples

$$1, 2, \cdots \in \text{Integer}$$
$$x^2,\ x^2 + 5y^2 \in \text{Polynomial(Integer)}$$

2.2 Domains

By a *domain* of computation, or simply domain, we mean:

- a set of generic operations.
- a representation.

- a set of functions which implement the operations in terms of the representation.
- a set of attributes which designate useful facts such as axioms and mathematical theorems which are true for the operations as implemented by the functions.

Examples of simple domains are those corresponding to the basic data-types offered by the system, for example Integer and String. We also may deal with domains, which are parameterised with one or more domains, they may also have items as parameters.

Examples

Polynomial(Integer), this domain corresponds to the space of polynomials with integer coefficients.

UnivariatePolynomial(x,Integer), this domain corresponds to the space of univariate polynomials with Integer coefficients, where the variable is the Symbol 'x'.

It is obvious that allowing the concept of parameterised domains may allow a computer algebra system to incorporate an infinite variety of Domains.

- The generic operations are given by the specification of an export list of signatures, the signatures are expressions consisting of an *operation name*, the *source domains* and a *target domain.*
  - The operation name is the name used to designate the function.
  - The source domain is a tuple consisting of the types of the parameters to the function.
  - The target domain is the type of the value returned by the function.
- The representation for a domain describes a data structure used to represent the objects of the domain.
- The functions part is a set of compiled functions which implement the operations in the export list.
- The attribute part of the domain is described either by a name e.g. "finite", to specify that the domain represents a finite set, or by a form with

operator names as parameters e.g. "distributive(*,+)", to specify that "+" is distributive over "*", ie. $(x + y) * z = x * z + y * z$.

2.3 Categories

A *category* designates a class of domains with common operations and attributes but with different functions and representations. In AXIOM a category heirarchy is built up starting at *BasicType*, which has one required operation, that of equality which must return a Boolean value.

To build the category heirarchy, new categories are built on top of old ones in the following manner:
A category may *inherit* operations or attributes from previous categories. They may also introduce their own operations and attributes. It is possible to define generic techniques to implement certain operations, in terms of other operations which **must** be implemented in a domain of this category. For example, in the category, *BasicType* $\tilde{}=$ is implemented by the code:

```
_~_=(x:\%,y:\%) : Boolean == not(x=y)
```

This function will automatically define the '~=' function in terms of the '=' function which has been defined by any domain claiming to be in that category, this idea constitutes a saving of effort on the part of the programmer, also a saving of space on the part of the computer.

2.4 Parameterised Categories

These are categories which are parametrised, the example we shall look at is:
MatrixCategory(R,Row,Col)
this is a domain where all the elements of the matrix are of the type 'R' which must be some Ring,

the domain parameters Row and Col must take values which are domains which are in the Category of FiniteLinearAggregate(R), for example List(R) or Vector(R).

Example

An example we might be interested in is the domain `Matrix(SLV(INT))` which is in the category `MatrixCategory(SLV(INT),Vector(SLV(INT)),Vector(SLV(INT)))`. Because we can then use cheap algorithms on these objects, for example calculating determinants of matrices of SLPs using Bareis method D.

Categories we are interested in are those with the algebraic property of GcdDomain (those domains in which there exists a gcd algorithm) and that of Euclidean Domain.

The following definition of Euclidean Domain we take from Pretzel [23].

Definition. 1 *let D be a domain. We shall call a function $||\cdot|| : x \to ||x||$ defined on the non-zero elements x of D with value in the non-negative integers a* Euclidean valuation *if:*

1) *For every a, $b \neq 0$ then $||ab|| \geq ||a||$ and $||ab|| \geq ||b||$.*

2) *For every a, $b \neq 0$ in D there exists a quotient q and remainder r in D, such that $a = qb + r$ and $||r|| < ||b||$ or $r = 0$.*

Definition. 2 *A* Euclidean Domain *is a domain in which there exists a euclidean valuation.*

2.5 Functors

By a *functor* we mean any function which returns a domain. A functor creates a domain, a member of some category. A functor creates a domain by storing functions into a template given by its target category.

Domains can only be built through functors. Basic domains can be built by functors bound to identifiers, for example the Integers, however more complex domains are built by functors which may take parameters, the parameters may be first order objects, for example Symbols, or they may be themselves domains. One functor which takes both a first order object and a domain as parameters is *UnivariatePolynomial.* The functor takes as parameters a Symbol and a domain which is a Commutative Ring. The Symbol corresponds to the variable in the polynomial and the coefficient domain corresponds to the Commutative Ring which must be supplied.

2.6 Mathematical Viewpoint

Definition. 3 *A Category* C *consists of two collections. One collection is known as the* objects *of C, or* Obj(C), *the other is known as the* arrows *of C, or* Arr(C). *For each arrow, f, there exists two associated objects, the source of f, called* source(f) *and the target of f, called* target(f).
The following rules must be satisfied also:
given a Category C:

$$(\forall g, f \in Arr(C)) \wedge (source(g) = target(f)) \Rightarrow$$
$$((\exists g \circ f \in Arr(C)) \wedge (source(g \circ f) = source(f)) \wedge (target(g \circ f) = target(g)))$$

for every object c there exists a unique identity arrow on c, called id_c

$$\forall k, g, f \in Arr(C) \Rightarrow ((k \circ (g \circ f), (k \circ g) \circ f \in Arr(C)) \wedge (k \circ (g \circ f) = (k \circ g) \circ f))$$

$$\forall f \in Arr(C) \Rightarrow ((id_{target(f)} \circ f = f) \wedge (f \circ id_{source(f)} = f))$$

AXIOMs category structure is based upon this mathematical category theory. In AXIOMs category structure, the exported functions are analogous to arrows and domains are analogous to objects.

Chapter 3

Implementation of Straight Line Programs in AXIOM

We have seen that the basic structure for a Straight Line Program, used to represent polynomials, may be realised by a list of Straight Line Instructions, one of which is tagged as the return value. In order to implement SLPs we have created a set of Axiom Domains which we shall describe in the following.

3.1 The OpNode Domain

This domain contains four values to represent the four different operations which may comprise an SLP, *plus*, *minus*, *times* and *quotient*. OpNodes are represented by SmallIntegers, the simplest data type available, ie. we have the equivalent of a C *enum*.

3.2 The StraightLineInstruction Domain

Elements of this domain are of three different types:
constant nodes - these represent the constants in the polynomial, they are represented by elements of the base ring of the polynomial, paired with a list of

residues relative to the list of moduli of section 3.16.
input nodes - these represent the variables of the polynomial, they are represented as the symbols.
operation nodes - these represent operations in the evaluation tree, for the polynomial, they are represented by elements of *OpNode*, and two integers which should be taken as pointers to the instruction at the position in the program.

3.3 The StraightLineProgram Domain

Elements of this domain comprise a list of *Straight Line Instructions* 3.2 which are intended to be evaluated consecutively. The main operation implemented in this domain is the *merge* operation, which merges two or more SLPs together, so that repeated operations only appear once, i.e. one form of redundancy is removed.
We also store a label associated with the instruction list to identify the program.

3.4 The Eval Domain

This domain contains evaluation functions, the algorithms used are those of section 4.2 and 4.3. We have implemented three complete evaluation functions for:

- Division free SLPs, if a division node is encountered, an error is raised.
- Division free SLPs, if a division node is encountered, the value failed is returned.
- SLPs with division, the value returned is in the quotient field of R, where R is the base ring, i.e. Fraction(R).

A partial evaluation function has also been implemented.

3.5 The EvalMod Domain

This domain contains evaluation functions which work over modular fields. We have implemented the following functions:

- A complete evaluation function which evaluates the SLP over a number of modular fields, then combines these results using the chinese remainder theorem.
- A complete evaluation function which evaluates the SLP over a given modular field.
- A partial evaluation function.
- A function to set the residues which correspond to every constant node in an SLP.

3.6 The Construct Domain

This domain contains functions which construct SLPs given the sparse form representation. We describe the two algorithms implemented.

3.6.1 Naïve version

This algorithm splits the sparse form polynomial into a sum of power products, each power is formed by using an addition chain technique, see section 3.6.3, these are multiplied, the resulting products are then summed.

3.6.2 Factor technique

1 Factorise the sparse form polynomial, using AXIOMs sparse form factoriser, for each factor

1.1 break the factor into a sum of monomials
for each monomial

1.2.1 split the monomial into a product of powers of single variables
for each variable

1.2.2.1 do we have this variable

yes $\rightarrow$ add multiplication nodes to the multiplication sequence until the variable is raised to the required power using the addition chain techniques of section. 3.6.3

no $\rightarrow$ create input node corresponding to the variable followed by the multiplication sequence the creation of which has been described

1.2.2.2 make balanced binary product tree to represent the monomial

1.3 make balanced binary summation tree to represent the factor

1.4 make balanced binary product tree to represent each factor being raised to some power

2 make balanced binary product tree to represent the polynomial

We create balanced binary trees in order to limit the depth of the SLP being created.

3.6.3 Addition Chains

The following section has been mostly taken from Knuth [14]. We shall consider the problem of constructing an SLP which calculates the polynomial x^n. Due to the additive nature of exponentiation, this problem reduces to the problem of constructing an *addition chain for* n, that is a sequence of integers:

$$1 = a_0,\ a_1,\ \cdots,\ a_r = n$$

with the property that $a_i = a_j + a_k$, for some $k \leq j < i$.
Once this addition chain has been calculated, we may create an SLP of the following form:

$$
\begin{array}{lll}
\text{line } \iota_0 : \text{InputNode} & : \text{x} \\
\vdots & \\
\text{line } \iota_i : \text{OperationNode} & : (\text{times line } \iota_j, \text{ line } \iota_k) \\
\vdots &
\end{array}
$$

where the line ι_r returns the value x^n.

binary method

We first consider a class of addition chains called binary addition chains, which are perhaps the most intuitive of addition chains. Some mathematicians believed that these were optimal (shortest) addition chains, however we shall see an example, where this is not the case.
The algorithm for creating binary addition chains follows:

1 Initialise: set Y to 1; set the list to $[Y]$

2 while $Y < n$ repeat

2.1 find the largest element on the list X s.t. $Y + X \leq n$

2.2 append $Y + X$ to the list set $Y \leftarrow Y + X$

3 return the list thus being formed

We consider an example of an SLP,created using the binary method to represent the polynomial which in sparse representation is x^6. The relevant addition chain is:

$$1,\ 2 = 1 + 1,\ 4 = 2 + 2,\ 6 = 4 + 2$$

The corresponding set of powers of x are:

$$x,\ x^2 = x * x,\ x^4 = x^2x^2,\ x^6 = x^4x^2$$

and the corresponding SLP is:

line 1 : Input node : x
line 2 : Operation node : (times line 1, line 1)
line 3 : Operation node : (times line 2, line 2)
line 4 : Operation node : (times line 3, line 2) >

Factor method

As promised we shall demonstrate a case where the binary method does not return the smallest possible addition chain. The smallest value for n where a smaller addition chain may be constructed by another means is when $n = 15$. The binary method would produce the chain:

$$1,\ 2 = 1 + 1,\ 4 = 2 + 2,\ 8 = 4 + 4,\ 12 = 8 + 4,\ 14 = 12 + 2,\ 15 = 14 + 1$$

which has 7 members, however smaller addition chains exist, with only 6 members, for example:

$$1,\ 2 = 1 + 1,\ 4 = 2 + 2,\ 5 = 4 + 1,\ 10 = 5 + 5,\ 15 = 10 + 5$$

We notice that this latter chain, in essence calculates $m = 5$ followed by $3m = 15$. This is an example of a chain constructed using the factor method of chain creation. If $n = pq$ where p is the smallest prime factor of n and $q > 1$. Then we may proceed to calculate x^n by first calculating x^p, and then raising this quantity to the qth power. If n is prime, we may caculate x^{n-1} and multiply by x. For example, we shall consider an addition chain which calculates $31 = 30 + 1 = (6 * 5) + 1$, which is prime:

$1,\ 2 = 1{+}1,\ 4 = 2{+}2,\ 6 = 4{+}2,\ 12 = 6{+}6,\ 24 = 12{+}12,\ 30 = 24{+}6,\ 31 = 30{+}1$

However sometimes it is best to just use the binary method in this case. Repeated application of these rules will return x^n.
If we allow division nodes in our representation we may utilise *addition-subtraction* chains, that is a sequence of integers:

$$1 = a_0,\ a_1,\ \cdots,\ a_r = n$$

with the property that $a_i = a_j + a_k \vee a_i = a_j - a_k$, for some $k \leq j < i$.
The element of the chain $a_i = a_j - a_k$ will correspond to the operation :

$$\text{line } \iota_i : \text{ OperationNode : (divide line } \iota_j, \text{ line } \iota_k)$$

we display an addition-subtraction chain to calculate $31 = 32 - 1$ which only has seven elements, this can be compared to the eight elements of the previous addition chain:

$$1,\ 2 = 1 + 1,\ 4 = 2 + 2,\ 8 = 4 + 4,\ 16 = 8 + 8,\ 32 = 16 + 16, 31 = 32 - 1$$

3.7 The ConstructCompileEval Domain

This domain holds functions which allow a special type of evaluation which is expounded in greater detail in section 4.5. The domain includes:

- Construction programs which construct Aldor [Appendix A] and Lisp [Appendix B] programs.
- Functions to compile and load the constructed programs.
- Functions to evaluate an SLP, using the compiled programs.

3.8 The SLPBound Domain

This domain contains functions to calculate bounds for an SLP evaluation, also functions to provide bounds for coefficients. The bounds that we require are all based on initially finding bounds to the coefficients, which we find by evaluating the polynomial at the point $(1, \cdots, 1)$ using a form of evaluation where all constants are replaced by their absolute value, also all subtractions are replaced by addions and all quotients replaced by multiplications. A faster 'evaluation' technique that we discovered which also finds a bound to the evaluation is described in 4.8.

3.9 The Compression Domain

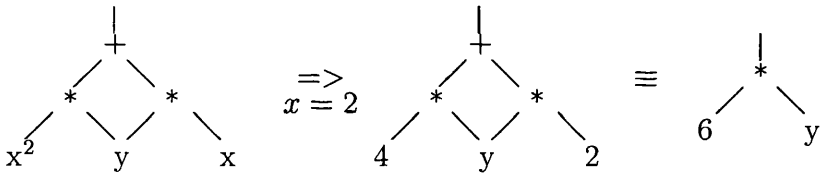
The simplest form of compression we implement is to remove any repeated operations, in a similar way to the merge operation given in section 4.4. Other optimisations we implement make use of the identities:

$1.x = x$

$x.0 = 0$

$x + 0 = x$

We consider a fundamental problem which compression techniques attempt to deal with. Consider the following arithmetic network and its evaluation at $x = 2$:



$x^2y + xy$ $\qquad$ $4y + 2y$ $\qquad$ $6y$

Of course the final form is the form we would prefer, however the resolution of this problem is far from straight-forward. In the next section we shall describe various compression techniques and propose a new resolution to the problem.

3.10 Traditional Techniques

Some work has been done on compression of SLPs by Nathalie Revol during her PhD work [26][27] extending the work of Miller, Ramachandran and Kaltofen [19]. We shall briefly describe this below.

The method consists of the repeated application of a procedure they term a *phase*. Each phase consists of three procedures, termed *MM*, $Eval_+$ and *Shunt* by Miller, Ramachandran and Kaltofen (generalizations of these procedures are

termed respectivly Group, Eval and PartialEval by Revol).
We shall now discuss the operation of these different steps:

3.10.1 MM

This operation groups together plus nodes by performing products on *Connection Matrices*, that is matrices which specify the connectivity of the SLP.

3.10.2 Eval_+

This operation compresses all plus nodes where both children are constant leaves into one constant node.

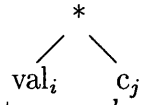
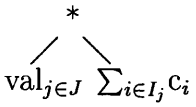
3.10.3 Shunt

This operation compresses all multiplication nodes in a way analogous to Eval_+. It then compresses trees of the form $c_1x + c_2x$ where x is an arbitrary node and c_1, c_2 are arbitrary constant nodes, to trees of the form $(c_1 + c_2)x$, where the operation $(c_1 + c_2)$ has been performed contracting it to a single node.

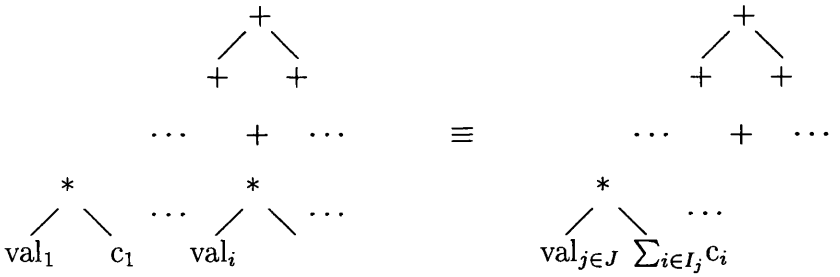
3.11 operation-Node Envelopes

In order to perform our compression, we have designed a compression technique, which we believe to be effective for SLPs. First we must make some definitions, which are important for our method:
Definition:
We define a **plus, minus, times or quotient envelope** to be a section of the DAG, made up of contiguous plus (respectively minus, times or quotient) nodes, and moreover any node may only be referenced by one other node in a given envelope.

We shall first consider **plus envelopes**:

The technique works by considering the nodes on the periphery of the envelope. If we encounter nodes of the form,  where c_j is a constant node, we may then match these periphery subtrees val_i. For every one of these periphery subtrees we find any other equal subtrees on the same periphery, we may form a partition based on this, for every envelope. We may now form an index, with a representative element from each partition. We call this index J. We now sum the constants in each partition. We are now able to replace each partition by a sum of the following nodes: 

in other words, we make the following compression:



The above compression makes use of the distributive rule:

$$ac_1 + ac_2 = a(c_1 + c_2)$$

The associative rules for plus and multiplication:

$$(ac_1)c_2 = a(c_1c_2)$$

$$(a + c_1) + c_2 = a + (c_1 + c_2)$$

may be used on times envelopes and plus envelopes respectively.

3.12 The CoeffConstruct Domain

This domain contains functions to construct SLPs given a list of SLPs which represent the coefficients. A subset of these polynomials are univariate polynomials, we have a function specifically to construct these given values from the Ring.

3.13 The StraightLineValue Domain

Elements of this domain are intended to represent polynomials. The representation consists of a pointer to a *Straight-Line Program* and a *Straight-Line Instruction* within that program. Therefore it becomes possible to represent many polynomials, using the same Straight-Line Programs, but different return instructions. This ability of representing different polynomials with the same program is demonstrated for example in the interpolation algorithm of section 5.5.
Other quantities that we store for each Straight-Line Value are:

- A degree list, this is a list of the exact degrees for each variable, stored as a union of degree and *failed* for the case that the degree is not known yet, or we only have an upper bound to the degree.
- A total degree, the total degree of the system stored as a union of degree and *failed* with the same meaning as for the degree list.
- A division free flag to record whether the system is division free or not.

These quantities are stored as they may take some time to calculate, for example, the degree list would require a complete interpolation for each variable, followed by possibly many evaluations to check whether the leading coefficients were zero or not.
We shall describe some of the operations that we have implemented for this domain. First we consider that the domain is intended to be in the category *CommutativeRing*, that is Rings with a multiplicative identity where multiplication is commutative. To satisfy this condition, it is necessary to export various

functions, for example:

- Functions which implement the arithmetic operations, $*$, $+$, $-$, $**$
- Functions to return the multiplicative and additive identities, 0 , 1
- A function to implement the equality operator, $=$.

We also implement various other functions, which will be required by the gcd function, we intend to implement.
The first we describe are a set of functions, with very similar implementations. That is, the function makes a call, starting at the return node of the SLP then descending the DAG, the recursion terminating at the leaf nodes. On the return phase a value (dependant on the function) will be returned, this may be altered at each level of the recursion. Many of the nodes in the SLP may be accessed multiple times. This allows us to make an optimisation: we store a global array of boolean flags, which say whether a node has been accessed before or not. This saves us from repeatedly calculating the same value. These functions are:

- The function depth:
  At a leaf node, a depth value of one will be returned.
  At an internal node, $node_i$, a call will be made on each of the nodes pointed to by $node_i$. The maximum of the return values plus one will be returned. This returns the depth of an SLP.
- The function slp2sparse:
  This returns a value from the domain Polynomial(R). This domain holds values which are polynomials with a base ring denoted by R.

  1) The node is an input node; this may be thought of as a monic univariate polynomial. This is the value returned.

  2) The node is a constant node; this may be thought of as a constant polynomial. This is the value returned.

  3) The node is an operation node;
  if the node is Operation node : ($\circ$: line i, line j), where $\circ$ is the instruction

The function makes calls on the nodes corresponding to line i and line j, the polynomials returned by these calls we shall denote p_i and p_j respectivly. The return value will be the polynomial $p_i \circ p_j$.

The final value returned is the value which corresponds to the return node of the SLP.

We note that for large SLPs this is an inefficient method. It would be more efficient to combine the technique of Zippel (pg. 242 of Effective Polynomial Computation[30]) with the interpolation method of section 5.5 taken from Kaltofen [11].

Example

An example to show how this function would operate on an SLP which represents the polynomial $x^2 + 2x$:

instruction list	returned polynomial
line 1 : Input node : x	x
line 2 : Constant node : 2	2
line 3 : Operation node : (times line 1, line 1)	x^2
line 4 : Operation node : (times line 2, line 1)	$2x$
line 5 : Operation node : (plus line 3, line 4) >	$x^2 + 2x$

- The function variables?
  This returns a list of the variables which appear in the SLP.

- The function slpDegree
  This returns a bound on the degree of the polynomial. We may not claim that it is equal to the degree of the polynomial, as we may get cancellation of leading coefficients, as shown in the following example.

  Example

  We may represent the polynomial which in sparse form is $2x + 1$, using the following SLP:

  line 1 : ConstantNode : 1
  line 2 : Input node : x
  line 3 : ConstantNode : 2
  line 4 : Operation node : (times line 2, line 3)
  line 5 : Operation node : (times line 2, line 2)
  line 6 : ConstantNode : 3

```
line 7 :  Operation node :  (times line 5, line 6)
line 8 :  Operation node :  (plus line 1, line 4)
line 9 :  Operation node :  (plus line 7, line 8)
line 10:  Operation node :  (minus line 9, line 7)   >
```

This is in essence a coding for the expression $3x^2 + 2x + 1 - 3x^2$: the slpDegree is two, however the degree of the polynomial is only one.

- The function slpTotalDegree
  This returns a bound on the total-degree of the polynomial. We may not claim it is equal to the total-degree, the same reason as for slpDegree applies. It is useful for our gcd algorithms, as it forms a bound on the degree of the leading variable in the translated polynomials which occur in the gcd algorithm of chapter 6.

In addition we export

- construction functions to allow us to build the objects.
- access functions:
  - instruction? which returns the return instruction of the SLP
  - program? which returns the program of the SLP

  and the corresponding functions to set these quantities
- evaluation functions to perform full and partial evaluation, over both modular and non-modular rings, for details see chapter 4.
- some equality based functions:
  - a zeroTest? function, this requires the set of moduli (of section 3.16 to be set. It performs evaluations over each modular field, however a Chinese Remainder step (detailed in appendix C) is not necessary. This is useful for equality checking.
  - eqConstant which tests whether an SLP is constant, this is required to check that the translation stage of the gcd algorithm in chapter 6 has resulted in a constant leading coefficient. This is a required property for our algorithm to work..

– various exact degree functions, these need to find the last non-zero coefficient of a polynomial.

3.14 Interpolation Domains

by interpolation we mean the determination of the coefficients of a polynomial. We have implemented functions to perform Lagrange interpolation for multivariate *StraightLineValues*. This technique is detailed in section 5.2. Also we have implemented the seperate technique detailed in section 5.5, which relies more heavily on the specific structure of SLPs.

3.15 The Gcd Domain

This domain contains functions $gcd(slp_1, slp_2)$ and $exactQuo(slp_1, slp_2)$ required to make it a GcdDomain. The gcd code uses a Las-Vegas version of the Monte Carlo gcd algorithm given by Kaltofen in [11].

3.16 An improved version

After producing code to implement the Gcd domain mentioned above, it became apparent that due to the modular setting used to reduce the cost of the arithmetic necessary, much redundant copying of programs was taking place. The reason was that in different moduli the operation and input nodes are not affected. This consideration requires changes in the representation at the StraightLineInstruction level of the SLP hierarchy. We shall consider the changes necessary to accomodate this problem. **Constant nodes**
With the old system a constant node would consist of simply a value from the underlying base Ring. If we require to work over a sequence of modular fields we require to store a set of SLPs, the base ring being the respective modular field.
In our improved version, every constant node is stored as an element of the base ring paired with an ordered set of residues, in each of a set of moduli. If we

require to increase the number of moduli, we must add to the set of moduli, and update the set of residues, for every constant node.

Pointers to the Constant nodes
When adding a modulus to the set of moduli for an SLP, it is necessary to update the sets of residues for each constant node. In order to remove the requirement of traversing the entire SLP, in order to find the constant nodes, we may store a set of pointers together with each SLP, pointing to the constant nodes. We may then access the node directly in order to calculate and store the new residue.

3.17 The Monte Carlo Domains

We consider the problems that occur when we attempt to embed a Monte Carlo domain, that is a domain which includes Monte Carlo algorithms (for example the equality checking algorithm detailed in section 4.14), in the AXIOM category heirarchy. Namely that the heirarchy and algorithms therein are not designed with Monte Carlo algorithms in mind. In other words, consider a domain which contains algorithms which call some Monte Carlo algorithms from some domain lower down the category heirarchy. This algorithm will have been designed with the expectation that the results that it obtains from any algorithm will be deterministically correct, clearly not possible if the algorithm is Monte Carlo (by definition). It would be necessary to make certain accomodations in order to implement such an embedding.

Definition 3.1 *We define the* pre-emptive probability of incorrectness *of objects in a Monte Carlo domain, as the probability of incorrectness of the arguments to an algorithm.*

The pre-emptive probabilities of incorrectness may be combined to give a lower bound to the probability of incorrectness of the value returned by an algorithm.

Definition 3.2 *We define the* implicit probability of incorrectness *as the extra*

probability of incorrectness added by a specific algorithm. The implicit probability of incorrectness of an algorithm together with the pre-emptive probabilities of incorrectness of the arguments may be combined to give the final probability that the answer is incorrect.

Firstly consider AXIOMs system of inheriting commands from a domain, this would require that the probability of correctness be stored together with the object. Now if we perform some Monte Carlo algorithm on two probabilistic objects where the probability that these objects are incorrect is p_1 and p_2, then the maximum probability that the answer returned is incorrect will be the probability of incorrectness if the two events are independent of each other. We justify this statement using the following argument: A bound on the probability of the correctness of two events is the sum of the probabilities. However this does not take into account any dependance or otherwise of the two events. We argue that in our case if there is any dependance between the two events then the probability of correctness of one event will in fact increase the probability of correctness of the other. Therefore an upper bound to the probability of incorrectness may be arrived at by assuming that the two events are independant. This may be calculated using the following formula. $p_1 + p_2 - p_1p_2$. This is the pre-emptive probability defined above. Any Monte Carlo algorithm will also introduce a further implicit probability, we denote this p_{imp}. Again we may find an upper bound by assuming that the two events are independant, we thus arrive at the expression:

$$p_1 + p_2 - p_1p_2 + p_{imp} - p_{imp}(p_1 + p_2 - p_1p_2) \tag{3.1}$$

3.17.1 Example

We consider the problem of determining whether $f_1(x, y) = f_2(x, y)$ where $\deg_x(f_1(x, y)) = 10$, $\deg_y(f_1(x, y)) = 15$ and $\deg_x(f_2(x, y)) = 15$, $\deg_y(f_2(x, y)) = 10$
Initially we consider the problem where the probabilities that f_1 and f_2 are incorrect is zero. This means that they have been derived exclusivly from deterministic algorithms.
The equality test we shall perform will involve ten evaluations. In the following we calculate the probability that we get an incorrect answer.
We shall work in a space of size $\rho \times \rho$, where $\rho > 2^{15}$. We may do this if we do

arithmetic mod ρ. A good value for ρ is 65521, this is the smallest prime $< 2^{16}$; i.e. can be stored in one 16 bit word.
maximum number of zeros = max = $65521 * 15 * 2 - 15^2 = 1965405$
number of points = pts = $65521^2 = 4294967296$
probability of one evaluation point being a zero whilst $f_1(x,y) - f_2(x,y)$ is not identically zero = $\frac{\text{max}}{\text{pts}} = \frac{1965405}{4294967296}$
probability of ten evaluation points being zero whilst $f_1(x,y) - f_2(x,y)$ is not identically zero = $\left(\frac{\text{max}}{\text{pts}}\right)^{10} = \left(\frac{1965405}{4294967296}\right)^{10} \approx 4 * 10^{-34}$
this is the propability of the answer being incorrect

Now we shall consider the same problem where the probabilities of incorrectness of f_1 and f_2 are 2^{-16}, 2^{-24} respectivly.
The smallest probability that we may return an incorrect answer is :

$$2^{-16} + 2^{-24} - 2^{-16-24} = \frac{2^{24} + 2^{16} + 1}{2^{40}}$$

if we substitute this and the inherent probability calculated above into equation 3.1 we obtain the following value

$$\frac{2^{24} + 2^{16} + 1}{2^{40}} + \left(\frac{1965405}{4294967296}\right)^{10} - \left(\frac{1965405}{4294967296}\right)^{10} * \frac{2^{24} + 2^{16} + 1}{2^{40}}$$

$$\approx 1.5 * 10^{-5}$$

The problem that arises in AXIOM is that it is not always possible to achieve a given absolute probability. This is because the preemptive probability of incorrectness of the arguments limit the probability of correctness of the answer. The best we may hope to achieve is to limit the implicit probability of incorrectness.

3.17.2 How we may embed this idea in AXIOM

We shall now consider how it would be possible to achieve an embedding of a Monte Carlo gcd domain in AXIOM. We must make sure that as an attribute, the domain has the category of GcdDomain. We prepose that a global variable may be defined in the Monte Carlo domain, which determines the inherent probability cost induced by any gcd command. If we use the method suggested by Kaltofen

[11] this will be $2P_{eq} + P_{eq}^2$ where P_{eq} is the probability allowed for an incorrect result from an equality test, see section 6.5. We would allow the user to alter this value, by exporting a function which could alter the global variable which specifies the implicit probability allowed for an incorrect answer. Alternativly we could specify that the implicit probability was a parameter for the domain.

Chapter 4

Evaluation strategies,

4.1 Introduction

In this chapter we describe evaluation strategies. In section 4.2 we talk about *complete evaluation* of an SLP, that is the specialisation of every variable in the program. Complete evaluation is necessary during equality checking, which we may reduce to zero checking of the difference of the two SLPs. We discuss different methods of equality checking in greater detail in sections 4.2, 4.12, 4.13 and 4.14. Complete evaluation is also necessary during checking whether an SLP is a constant polynomial. In section 4.3 we discuss *partial evaluation* of an SLP, that is the specialisation of a subset of the variables in a program. In section 4.4 we discuss the methods used for *merging* SLPs, this is the process used to remove redundant (repeated) operations when combining programs. In section 4.5 we discuss a method of evaluation which constructs *Aldor* or *Lisp* programs to perform the evaluation, which may be compiled before being executed. In section 4.8 we discuss a fast method of calculating bounds to the size of an evaluation. In section 4.14 we discuss a classic Monte Carlo technique for equality checking due to Schwartz [10].

4.2 Complete Evaluation of a Straight Line Program

The technique used to perform complete evaluation is a fairly straight-forward one:

1. set up an array, to hold intermediate values from the base field, denote the array A
2. iterate through each node in SLP, at each step denote the current node by *node*:
   (a) if *node* is a constant node, set the corresponding element of A to this constant,
   (b) if *node* is an input node, set the corresponding element of A to the required specialisation,
   (c) if *node* is an operation node, perform the corresponding operation on the elements of A which corresponds to the nodes pointed to by the pointers of *node*. Finally store the result in the element of A which corresponds to *node*.

If an input node has not been given a value, this is an error, as the evaluation is not a complete evaluation. Also if the elements of A corresponding to the pointers of an operation node have not been set, then this is an error. A constant from the base field will be returned. For efficiency we may evaluate at a number of small prime moduli, then use the *chinese remainder theorem* (CRT) (see appendix [C]) to find the constant to return.

4.3 Partial evaluation of a Straight LineProgram

The technique used to implement partial evaluation of an SLP is far more complex than that for the complete evaluation as the return value is in fact an SLP

which must be constructed, a far from simple operation, as any resultant shrinking of the program will result in pointers being changed. This must be correctly reflected in the result.
As the following example shows, the result may be far from optimal. It would be a good idea to use a compression technique to improve the optimality.

Example

We shall consider the example of the program which represents the polynomial $x^2y + xy$, that is:

```
line 1 :  Input node      : y
line 2 :  Input node      : x
line 3 :  Operation node  : (times , line 2 , line 2)
line 4 :  Operation node  : (times , line 3 , line 1)
line 5 :  Operation node  : (times , line 2 , line 1)
line 6 :  Operation node  : (plus , line 4 , line 5)   >
```

If we perform the partial evaluation of this SLP at the point $x = 2$, i.e. we perform the specialisation $x = 2$, we get the SLP:

```
line 1 :  ConstantNode  : 4
line 2 :  ConstantNode  : 2
line 3 :  InputNode     : y
line 4 :  OperationNode : (times , line 1 , line 3)
line 5 :  OperationNode : (times , line 2 , line 3)
line 6 :  OperationNode : (plus , line 4 , line 5)   >
```

This represents the polynomial $6y$, which may be represented by the SLP:

```
line 1 :  ConstantNode  : 6
line 2 :  InputNode     : y
line 3 :  OperationNode : (times , line 1 , line 2)   >
```

This constitutes a saving of fifty percent in the program length.

Partial evaluation is necessary during the Lagrange interpolation and Fourier interpolation of multivariate SLPs, we shall discuss these techniques in sections 5.2 and 5.3.

4.4 Merging

4.4.1 Merging two SLPs

The basic technique that we shall use is the following:

Program $==$ `Merge`$_2$
input (`slp`$_1$, `slp`$_2$)

1. answer := the longest argument program.
2. Iterate through the instructions of the shortest program ,
   (a) check whether the current instruction is in the first program,
      i. if so we need to store the position, as the pointers in any operations which point to this instruction will need to know the new position.
      ii. if not and the instruction is an operation node,
      we may need to change the pointers, before we add it to answer

If the lengths of the two argument programs, slp_1 and slp_2, are l_1 and l_2 respectivly, we may assume WLOG. that $l_1 \geq l_2$. Each check takes $O(l_1)$ operations. As a check must be performed for each instruction in the second program, the merge operation takes $O(l_1 l_2)$ checks.
We note that step two involves locating two operations in the program already created, it is a good policy therefore to apply step one to the longest program.

4.4.2 An improved merge

If we first create a hash table containing the operations in slp_1, then each of the check operations will be cheaper.
Suppose the creation of one hash table element takes h_c basic operations, and each look up takes h_l basic operations.
Then the new merge operation takes t basic operations, where $h_c l_1 + 2h_l l_2 \leq t \leq h_l l_1 + (2h_l + h_c)l_2$.
h_l and h_c will be *almost* independant of l_1, (they depend on the number of *clashes* that occur during creation of the hash table). If we assume that no clashes occur then the merge operation takes $O(l_1 + l_2)$ basic operations. We effectively reduce the complexity from quadratic to linear.
A problem with this technique occurs if l_1 is much greater than l_2, in this case the creation of the hash table becomes the overriding factor. h_l has the same time complexity as h_c, therefore, it is again a good policy to take slp_1 as the longest program.

4.4.3 Merging of n SLPs

A straight forward technique would be to use the following approach:

Program == $\texttt{Merge}_n$
input ($\text{slp}_1 \cdots \text{slp}_n$)

1. merged := slp_1
2. iterate i=2 to n
   (a) merged = merge_2(merged,slp_i)

The problem with this algorithm is that the variable merged will accumulate the result. As pointed out above the creation of the hash table will become the overriding factor. We now consider the complexity for this operation:
The merge operation will result in a smaller SLP. In the following δ_i denotes the decrease in length of an argument SLP slp_i, during the creation of its image under

merge. $\tilde{l}$ denotes the arithmetic mean of l_i, $1 \leq i \leq n$, where l_i is the length of slp_i.

Consider the time-complexity of the hash table creation, denote this complexity C_h.

$C_h = h_c l_1 + h_c(l_1 + l_2 - \delta_2) + h_c(l_1 + l_2 + l_3 - \delta_2 - \delta_3) + \cdots + h_c(l_1 + \cdots + l_n - \delta_2 - \cdots - \delta_n)$

in the average case δ_i will be small so

$C_h \approx h_c(nl_1 + (n-1)l_2 + \cdots + l_n)$

$= O(n^2\tilde{l})$

Also consider the time-complexity for looking up an element in a hash table, denote this complexity

$C_l = h_l(l_2 + \cdots + l_n) = O(n\tilde{l})$

so the complexity of $\texttt{Merge}_n$ is $O(h_c n^2 \tilde{l}) + O(h_l n\tilde{l}) = O(n^2\tilde{l})$

4.4.4 An improved merge for n SLPs

In order to merge n SLPs, rather than iteratively using merge_2 on pairs of SLPs we create a routine specially designed for merging n-tuples of SLPs:

Program == $\texttt{BetterMerge}_n$

input $(\text{slp}_1 \cdots \text{slp}_n)$

1. put slp_1 on the hash table
2. iterate through slp_2 to slp_n
   (a) include slp_i (the program under consideration) in result
   (b) break if slp_n reached
   (c) put slp_i on the hash table

We now consider the complexity analysis:

complexity of the hash table creation $= h_c \sum_{i=1}^{(n-1)} l_i = (n-1)\tilde{l} = O(n\tilde{l})$

complexity of the hash table look up $= h_c \sum_{i=2}^{n} l_i = (n-1)\tilde{l} = O(n\tilde{l})$

so the complexity of $\texttt{BetterMerge}_n$ is $O(n\tilde{l})$ basic operations.

Once again for optimal performance the ordering of the SLPs with respect to their length is important. This ordering should be with the longest first, shortest last.

4.5 Evaluation via. Compiled SLPs

We consider one technique of creating an ALDOR (see appendix A) program, which we may then compile to give an executable program which on execution will return the (complete or partial) evaluation of a specific SLP. The execution of the program when compiled will be faster than the execution of the general technique described in section 4.2. This is analogous to compiled versus interpreted normal code. The compilation will take some time, so this strategy is only good if the evaluation must be done many times. We note that it would be theoretically possible to use machine code for the representation thus cutting out the need for the compilation step, however this would require a different implementation for each architecture. A compromise would be to create a Lisp program (see appendix B), as this would cut out one stage of compilation.
As with *interpreted* evaluations a different style of program must be created for complete and partial evaluations.
Because of the cost implicit in the compilation step, one technique may be to use the interpreted evaluation method if the evaluation is to be repeated $\leq n$ times (where n is the break even number of repetitions), or the compiled evaluation if the evaluation is to be repeated more than n times.
Example
polynomial given in sparse representation is:

$$p(x, y) = x^2y^2 + 1 \tag{4.1}$$

original SLP is:

```
                slp1
line 1 :  InputNode      :  x
line 2 :  OperationNode  :  (times , line 1 , line 1)
line 3 :  InputNode      :  y
line 4 :  OperationNode  :  (times , line 3 , line 3)
line 5 :  OperationNode  :  (times , line 2 , line 4)
line 6 :  ConstantNode   :  1
line 7 :  OperationNode  :  (plus , line 5 , line 6)   >
```

In the following, we shall assume that this SLP is being held in a variable called 'slp1'.
A program which we may create to perform complete evaluation of slp1 is:

```
temp(x:INT,y:INT):INT == {
  t1 := x;
  t2 := t1 * t1;
  t3 := y;
  t4 := t3 * t3;
  t5 := t2 * t4;
  t6 := 1;
  t7 := t5 + t6;
  t7
}
```

To construct such a program we could make a function call as follows:

```
exactEval(slp1,['x,'y])$ConstructCompileEval(INT)
```

After the compilation of this program, to perform an evaluation we call the function temp, for example:

```
temp(2,3);
```

will perform the evaluation of slp1 at $x = 2$, $y = 3$ we would get the answer:

```
>37
```

It became apparent that creating a Lisp program to perform the evaluation was worthwhile, as the compilation time becomes negligable and the subsequent evaluations are considerably faster than the application of the Aldor functions, for the above problem the Lisp created is:

```
(defun dummy (lr p)
; *** this is where we put the compiled SLP ***
(setq tmp (make-array 6))
(setf (aref tmp 0) (rem (aref lr 2) p))
(setf (aref tmp 1) (rem (* (aref tmp 0) (aref tmp 0)) p))
(setf (aref tmp 2) (rem (aref lr 1) p))
(setf (aref tmp 3) (rem (* (aref tmp 2) (aref tmp 2)) p))
(setf (aref tmp 4) (rem (* (aref tmp 3) (aref tmp 1)) p))
(setf (aref tmp 5) (rem 1 p))
(setf (aref tmp 6) (rem (+ (aref tmp 5) (aref tmp 4)) p))
(aref tmp 6)
```

Note that this will perform a modular evaluation, where the modulus is the argument p.

4.6 Evaluation of non-division free SLPs

We consider some of the problems encountered when evaluating non-division free SLPs. If we consider the method of section 4.2, on encountering a quotient node we must calculate in the field of fractions of the base ring. On subsequently encountering a return node we may retract to the base ring, as we are guaranteed that the value will be non-fractional. Failure of this retraction indicates that the SLP does not represent a polynomial. If we are evaluating over $\mathbf{Z}_p$ for a number of prime moduli p, we may find an inverse in the modular field, usng the extended

euclidean algorithm, Aldor includes a function expressly for that purpose. If we are using the method of section 4.5 then we must include a Lisp function which performs this algorithm, viz.:

```
;function to calculate the inverse of a value, in Z_p, use extended
;euclidean algorithm
(defun inv (x y)
  (if (zerop x) (return-from inv (+ y 1)))
  (setq ttmp y)
  (setq u 0)
  (setq v 1)
  (while (not (= x 1))
     (if (zerop x) (block-return inv (+ ttmp 1)))
     (setq d (divide y x))
     (setq tm x)
     (setq x (cadr d))
     (setq y tm)
     (setq tm v)
     (setq v (- u (* v (car d))))
     (setq u tm))
  (if (< v 0) (setq v (+ v ttmp)))
  v)
```

If an SLP is not division free there may be points at which a division by zero error may occur, this will be because a factor of the denominator which necessarily cancels with a factor of the numerator evaluates to zero (here the terms numerator and denominator refer to the polynomials represented by the SLPs corresponding to the operations pointed to by the left and right pointers of the quotient operation causing the error). We note that in the case of a division by zero error, `inv` will return $p+1$, where p is the prime modulus; this is so that any function calling `inv` may be aware of the error and not expect a correct result. An example of a line which corresponds to a division node, to be inserted in a Lisp function is the following; it will perform the call to inv, and in the case of an invalid return value will return $p+1$.

```
(setf (aref tmp 3) (rem (* (aref tmp 2) (setq i (inv (aref tmp 1) p)))
```

```
p)) (if (= i (+ p 1)) (block-return dummy (+ p 1)))
```

An example of a valuation which displays this sort of error is detailed in the following. Consider an SLP equivalent to the polynomial:

$$p(x) = \frac{x^2 - 1}{x - 1} = x + 1$$

viz.

```
                        slp1
line 1 :  ConstantNode  : 1
line 2 :  InputNode     : x
line 3 :  OperationNode : (times line 2, line 2)
line 4 :  OperationNode : (minus line 3, line 1)
line 5 :  OperationNode : (minus line 2, line 1)
line 6 :  OperationNode : (quotient line 4, line 5)   >
```

If we wish to evaluate $p(x)$ at $x = 1$, we might expect the result $p(x) = 2$, however a straight forward evaluation would fail with a division by zero error at line six. During interpolations or equality checks we could simply try a different point, however if we specifically require the evaluation at this point then a solution would be to remove the division causing the problem. For example by using the technique of section 5.7, or by using the (p-adic) L'Hôpital's rule. A *true* division by zero error would imply that the SLP represented a rational function, using the techniques suggested above we would enter an infinite loop in this case (we will not encounter this problem, as we are only concerned with SLPs which represent polynomials (sums of power products) where this will never occur).

4.7 Returning multiple values

We may return more than one value from an SLP. We see an example of this in the interpolation technique given in section 5.5. In an SLP every coefficient corresponds to a different value. We shall see that there are situations where

we would like to return more than just the evaluation of one of these values. One example is the calculation of the degree of a polynomial given in the mixed representation of section 5.2.1. To do this we could use the following algorithm.

1. evaluate the entire SLP at each of the points specified in theorem 4.2 section 4.11, return a vector of values which correspond to the evaluations of each coefficient at each of these values.
2. return the index corresponding to the highest coefficient which returns a non-zero answer

It is necessary to perform fast complete evaluations of functions as repeated evaluations of the same program is necessary especially during equality checking. The same is not true of partial evaluations, so we have only implemented the conversion to Aldor programs which we may then compile.

A program which we may create to perform the partial evaluation at $x = \xi$ (where ξ is an arbitrary point) of the SLP representing equation 4.1 is:

```
temp(x:INT):SLV(INT) == {
  import from OR,SLI(INT),SLP(INT);
  slp1:Symbol := ''slp1''::Symbol;
  t1 := x;
  t2 := t1 * t1;
  iList:List(SLI(INT)) := [
    construct(1),
    construct(t2),
    construct(y),
    construct(construct(times),3,3),
    construct(construct(times),2,4),
    construct(construct(plus),1,5)];
  inst:SLI(INT) := iList.6;
  construct(inst,construct(slp1,iList,[y]))
}
```

To construct such a program we could make a function call as follows:

```
incompleteEval(slp1,['x])$ConstructCompileEval(INT)
```

After the compilation we may form an SLP which represents a polynomial in y namely the polynomial given in 4.1 with x specialised to a given value. For example we could make the call:

```
temp(2);
```

this would return the SLP:

```
                    slp1
line 1 : ConstantNode  : 1
line 2 : ConstantNode  : 4
line 3 : InputNode     : y
line 4 : OperationNode : (times line 3, line 3)
line 5 : OperationNode : (times line 2, line 4)
line 6 : OperationNode : (plus line 1, line 5)   >
```

this represents the polynomial which in sparse form may be written $4y^2 + 1$, ie the evaluation of equation 4.1 at $x = 2$.

4.8 Bound calculation

We first recall some bounds which apply to polynomials over the integers that are relevent to our work. We take these from the literature:
A result due to Mignotte [18] (Theorem **4.4.4** in his work) states :
if $P = Q_1 \cdots Q_m = \sum a_{j_1 \cdots j_n} X_1^{j_1} \cdots X_n^{j_n}$ we have

$$L(Q_1) \cdots L(Q_m) \leq 2^{d_1 + \cdots + d_n} M(P) \leq 2^{d_1 + \cdots + d_n} ||P|| \tag{4.2}$$

in the previous $L(P)$ is the *length* of a polynomial, it must not be confused with the length of an SLP, which we define in section 1.4, it is defined as:

$$L(P) = \sum_{k=0}^{d} |a_{j_1 \cdots j_n}| \tag{4.3}$$

$M(P)$ is defined as the *measure* of a polynomial, we are not concerned with its definition, as from the previous we have:

$$L(Q_1) \cdots L(Q_m) \leq 2^{d_1 + \cdots + d_n} ||P|| \tag{4.4}$$

d_i is defined as the degree of P in the variable X_i.
$||P||$ is defined as:

$$||P|| = (\sum |a_{a_{j_1} \cdots j_n}|^2)^{1/2}$$

From this we may ascertain bounds which apply to the gcd of polynomials.

We propose two schemes for calculating bounds for the coefficients of a polynomial represented as an SLP. The first scheme is for division-free SLPs, which are over the integers. It is a very fast technique and allows us to calculate a bound that enables the use of the chinese remainder theorem. The second technique is for SLPs which may contain division nodes, they may also be over the rationals. It is also a fast technique, though will take roughly twice the time of the first technique (this is not surprising as the base field has been squared). It also allows a bound to be calculated which enables the conversion described in appendix [C]. The basic idea is to do arithmetic to one significant figure (we shall use a binary base, as we are not concerned with the human readability of these bounds). We always round up and we work in absolute value. The algorithm for division free SLPs follows:

The idea is to build an array of values from the base field where each element of the array corresponds to a line in the program. Each value will be a bound to the evaluation of the program, up to the corresponding line, at the point $(1, \cdots, 1)$. The final bound to the evaluation at this point will be a bound to any coefficient in the polynomial assuming every coefficient is positive. This may be achieved by a prior transformation as well as doubling the bound to accomodate the negative part of the ring.

1. For every line in the program.

(a) If the line is an input node store one in the corresponding array element.

(b) If the line is a constant node store 2^{n+1} in the corresponding array element, where the constant node is either $a_n a_{n-1} \cdots a_0$ or $-a_n a_{n-1} \cdots a_0$

(c) If the line is an operation node, there are three possibilities:
We shall assume that the line is the following: operation line i, line j
where the value associated with line i is: $a = a_n a_{n-1} \cdots a_0$
and the value associated with line j is: $b = b_m b_{m-1} \cdots b_0$
assume WLOG that $a > b$,

i. the operation is plus or minus:
If $n = m$ then the corresponding value is 2^{n+2}.
If $n \neq m$, we may assume WLOG that $n > m$, the value required is $(a_n + 2)2^n$.

ii. the operation is times:
The value corresponding to this node is $2^{(n+m+2)}$

The method which works for all SLPs over $\mathbf{Q}$ relies on finding a pair of values which we will term a *rational bound.* This we define as a bound on the numerator and a bound on the denominator of the value. That is, given $q \in \mathbf{Q}$ s.t. $q = \frac{n}{d}$, find a pair of numbers $(\overline{n}, \overline{d}) \in \mathbf{Z} \times \mathbf{Z}$ with $\overline{n} > n$ and $\overline{d} > d$.
We will specifically look for $\overline{n} = 2^{\tilde{n}} > n$ and $\overline{d} = 2^{\tilde{d}} > d$.

1. For every line in the program.

(a) If the line is an input node, these will be specialised to $1 = \frac{2^0}{2^0}$ so we take $\tilde{n} = 0$ and $\tilde{d} = 0$.

(b) If the line is a constant node with value $c = \frac{c_n}{c_d}$, we consider the absolute value $|c| = \frac{|c_n|}{|c_d|}$. We may take $\tilde{n} = \lceil \log_2(|c_n|) \rceil$ and $\tilde{d} = \lceil \log_2(|c_d|) \rceil$.

(c) If the line is an operation node which on evaluation of the SLP at $(1, \cdots, 1)$takes the value $\frac{a}{b}$, there are three possibilities:
We shall denote the values taken by the left and right arguments as $\frac{a_l}{b_l}$ and $\frac{a_r}{b_r}$ respectively. We shall denote the rational bounds corresponding to $\frac{a_l}{b_l}$ and $\frac{a_r}{b_r}$ as (l_n, l_d) and (r_n, r_d) respectively and the corresponding exponent pairs $(\tilde{l_n}, \tilde{l_d})$ and $(\tilde{r_n}, \tilde{r_d})$ respectively.

i. The node is a times node:
Then $\frac{a}{b} = \frac{a_l}{b_l} \cdot \frac{a_r}{b_r}$.
So we need a bound on,
$a_l \cdot a_r \leq l_n \cdot r_n \leq 2^{(\tilde{l_n} + \tilde{r_n})}$ and $b_l \cdot b_r \leq l_d \cdot r_d \leq 2^{(\tilde{l_d} + \tilde{r_d})}$,
The exponent pair corresponding to this value is $(\tilde{l_n} + \tilde{r_n}, \tilde{l_d} + \tilde{r_d})$.

ii. The node is a division node:
Then $\frac{a}{b} = \frac{a_l}{b_l} / \frac{a_r}{b_r}$.
So we need a bound on,
$a_l \cdot b_r \leq l_n \cdot r_d \leq 2^{(\tilde{l_n} + \tilde{r_d})}$ and $b_l \cdot a_r \leq l_d \cdot r_n \leq 2^{(\tilde{l_d} + \tilde{r_n})}$,
The exponent pair corresponding to this node is $(\tilde{l_n} + \tilde{r_d}, \tilde{l_d} + \tilde{r_n})$.

iii. The node is a plus or minus node:
Then $\frac{a}{b} = \frac{a_l}{b_l} \stackrel{+}{-} \frac{a_r}{b_r} = \frac{a_l b_r \stackrel{+}{-} a_r b_l}{b_l b_r}$.
We treat all differences as summations, so we need a bound on,
$a_l b_r + a_r b_l \leq l_n r_d + r_n l_d \leq \max\{2^{(\tilde{l_n} + \tilde{r_d})}, 2^{(\tilde{r_n} + \tilde{l_d})}\}$
and $b_l b_r \leq l_d r_d \leq 2^{(\tilde{l_d} + \tilde{r_d})}$,
unless $\tilde{l_n} + \tilde{r_d} = \tilde{r_n} + \tilde{l_d}$ in which case
$a_l b_r + a_r b_l \leq l_n r_d + r_n l_d \leq 2^{(\tilde{l_n} + \tilde{r_d} + 1)}$.
The exponent pair corresponding to this node is $(\max\{\tilde{l_n} + \tilde{r_d}, \tilde{r_n} + \tilde{l_d}\}, \tilde{l_d} \tilde{r_d})$,
In the exceptional case the bound for the numerator must be $\tilde{l_n} + \tilde{r_d} + 1$.

At the end of the computation (that is when we get to a return node), we will have a rational bound for the evaluation of the SLP at the point $(1, \cdots, 1)$ of $(2^{\tilde{n_f}}, 2^{\tilde{d_f}})$. The modular field which we must work in must be greater than the product of these values. We note that this rational bound is not a bound to the absolute value of the evaluation, but bounds to the absolute value of the numerator and denominator of the value of the evaluation.

By performing this computation on the SLP in this way we find an upper bound to the coefficients. We may find a smaller yet still valid bound by:

1. finding a product of primes $\prod p_i$ such that $\prod p_i < 2B$, B being the upper bound arrived at by the previous evaluation; the factor of two allows for negative numbers.

2. performing evaluations of the SLP in each of the modular fields $\mathbf{Z}_{p_i}$

3. finally raising these modular results to a bound in $\mathbf{Z}$ using the Chinese Remainder Theorem (CRT).

It is important that we do-not evaluate the power of two, because this may actually be a large number, we would instead like to just calculate the exponent e, then use that to find a sufficiently large set of primes as follows:

1. for a set of primes $\{p_1, \cdots, p_i\}$
2. find a set $\{m_1, \cdots, m_i\}$ s.t. $2^{m_i} \leq p_i$
3. the set $\{p_1, \cdots, p_i\}$ is sufficient if $\sum_{1 \leq j \leq i} m_j \geq e$

We note that if we are simply holding the exponent we cannnot find a smaller bound by using the CRT.
We may also use the above technique for finding a bound to the evaluation at a specific point.

4.9 Equality testing via SLP evaluations

Due to the fact that SLP representation is a non-canonical representation, polynomial equality testing becomes a non-trivial task. In order to check the equality of two polynomials represented by SLPs, SLP_1 and SLP_2, we consider the difference SLP representing the difference between the two polynomials, we denote this by $d(x_1, \cdots, x_n)$.We evaluate $d(x_1, \cdots, x_n)$ at enough points so that if we get zero evaluations at every point, we are guaranteed that $d(x_1, \cdots, x_n)$ is identically zero, ie. that $SLP_1 = SLP_2$. We first consider the one dimensional case, namely is $SLP_1(x) = SLP_2(x)$ where x is a scalar variable. Information we have about SLP_1 and SLP_2 is the SLPdegrees (see section 3.13) of the SLP in the variable x, these are in fact bounds for the actual degrees of the polynomials represented by SLP_1 and SLP_2 in x. The maximum of the two SLPdegrees, will form a bound for the degree of x in the polynomial $d(x)$. We denote the maximum by m.

4.10 One Dimensional case

To check whether $p(x) = q(x)$ we may solve the equivalent problem $p(x) - q(x) = 0$. We shall denote $p(x) - q(x)$ by $d(x)$, also the upper bound for the degree of $d(x)$ in x by $\deg_x$. We perform evaluations of $d(x)$ at each of the integers in the closed interval $[0, \deg_x]$.

Theorem 4.1 (Univariate zero test)
If a polynomial $p(x)$ evaluates to zero at at least $\deg_x(p(x)) + 1$ points then $p(x) = 0$
proof :
Base case: *If $\deg_x(p(x)) = 0$ then if $p(x)$ evaluates to 0 at at least one point, since it is a constant polynomial $p(x) = 0$.*
Inductive Case: *Assume as an inductive hypothesis that the theorem is true for a polynomial q s.t. $\deg_x(q) = n$.*
If a polynomial $p(x)$, with $\deg_x(p(x)) = n + 1$ evaluates to zero at $n + 2$ points, then by Rolle's Theorem, there exists at least one point between each zero, s.t. $\frac{dp(x)}{dx}$ evaluates to zero at these points, ie. at at least $n + 1$ points.
$\frac{dp(x)}{dx}$ has degree at most n, so by our inductive hypothesis, $\frac{dp(x)}{dx} = 0$.
Now $\frac{dp(x)}{dx} = 0 \Rightarrow p(x) = c$ for some constant c, but $p(x)$ evaluates to zero at at least one point,
$\therefore\ p(x) = 0$

□

4.11 n Dimensional case

Theorem 4.2 *if $d(x_1, \cdots, x_n) = p(x_1, \cdots, x_n) - q(x_1, \cdots, x_n)$ evaluates to zero at every point in $X_1 \times \cdots \times X_n$ where $X_i = \{\tilde{x}_i | i \in [0 \cdots \max\{\deg_{x_i}(p), \deg_{x_i}(q)\}] \wedge \tilde{x}_i \in \mathbf{Z}\}$*
then
$p(x_1, \cdots, x_n) = q(x_1, \cdots, x_n)$
proof :
Base case: *The Univariate zero test 4.1 may be used as a basis for induction.*

Inductive case: *Assume,*

$$p(x_1, \cdots, x_i, \tilde{x}_{(i+1,j)}, \cdots, \tilde{x}_{(n,j)}) = q(x_1, \cdots, x_i, \tilde{x}_{(i+1,j)}, \cdots, \tilde{x}_{(n,j)})$$

or equivalently,

$$d(x_1, \cdots, x_i, \tilde{x}_{(i+1,j)}, \cdots, \tilde{x}_{(n,j)}) = 0$$

where $\tilde{x}_{(i,j)} \in \mathbf{Z}$
we have $d(x_1, \cdots, x_i, \tilde{x}_{(i+1,j)}, \cdots, \tilde{x}_{(n,j)}) = 0$ *for* $deg_j(d)+1$ *different values* $\tilde{x}_{(i+1,j)} \in \mathbf{Z}$.
now consider some arbitrary point $\xi = (\xi_1, \cdots, \xi_{i+1}) \in \mathbf{Z}^{i+1}$
we know that $d_{proj}(\xi_1, \cdots, \xi_i, \tilde{x}_{(i+1,j)}) = 0 \mid 0 \leq j \leq deg_j(p)$
$\therefore$ *the univariate polynomial* $d_{proj}(\xi_1, \cdots, \xi_i, x) = 0$ *by univariate zero test.*
$\therefore$ $p_{proj}(\xi) = 0$ *where* p_{proj} *is the projection of* p *onto* $\mathbf{R}^{i+1}$
$\therefore$ *by induction* $d(x_1, \cdots, x_n) = 0$
$\therefore$ $p(x_1, \cdots, x_n) = q(x_1, \cdots, x_n)$

□

The problem with the above method is that the number of points at which evaluations must be performed is exponential in the number of variables. In fact the number of points at which we must evaluate is $\prod(d_{x_1} + 1) \cdots (d_{x_n} + 1)$. This is clearly not a good state of affairs, especially when there are a lot of variables, we shall consider three other methods in sections 4.12, 4.13 and 4.14.

4.12 Small point method

In this method of zero testing we test for a zero evaluation at one point, if this evaluation gives zero we evaluate at another point so close that the only way in which there could be another zero evaluation is if the polynomial was identically zero. In the univariate case we shall use a bound on the separation of roots, this is due to Mignotte **Mignotte**[18] (proposition 6.4.10 in his work); we shall denote this by B in the following, whilst in the multivariate case we see that (modulus conjecture 4.1) after some manipulation we may also use this bound.
According to Mignotte, for the polynomial f:

$$B = d^{-(d+2)/2}.||f||^{1-d} \tag{4.5}$$

where d is the degree of f.

Theorem 4.3 (Univariate Small Point Zero test)
given $f \in K[x]$, *with* $f(x_1) = 0$, *and* $f(x_1 + \varepsilon) = 0$; *where* $0 < \varepsilon < B$, *then* $f(x) = 0$.
proof :
The proof is clear, see [18]

Conjecture 4.1 (Multivariate zero test)
given $f(x_1, \cdots, x_n) \in Q[x_1, \cdots, x_n]$, *where* $f(x_1, \cdots, x_n)$ *evaluates to zero at each corner of the* n *dimensional box, which has side* i $|1 \leq i \leq n$ *of length* $\varepsilon_i = \frac{bound(\varepsilon_1, \cdots, \varepsilon_{i-1}, x_i, 0_{n-i})}{2}$ *then* $f(x_1, \cdots, x_n)$ *is identically zero, where* ε_1 *is the Mignotte bound 4.5 applied to the polynomial* $f(x_1, 0_{n-1})$.

Asymptotics:
Clearly this technique involves evaluation at 2^n points, where the points become exponentially small.

We have decided not to consider this technique any further as the number of points is exponential in the number of variables, also roots of polynomials can be very close as the following example demonstrates. Consider the following 'simple' polynomials $x^3 - 5$ and $x^2 - 3$, working to two decimal places, these have roots 1.73 and 1.70 respectively, these are only 0.03 apart, if we were to ask the question is $(x^3 - 5)(x^2 - 3) = 0$ using the formula 4.5 we would allready have to consider points smaller than $(O(10^8))^{-1}$.

4.13 Very large evaluation point method

In this method of zero testing, we see that the problem may be solved by evaluating the polynomial at one very large point. In essence we show that if the evaluation comes to zero at one point, then either the polynomial is identically zero, or a factor exists which must have a coefficient which is higher than a certain upper limit for coefficients of any factor. Clearly the first option is the only possibility. However it is expensive to perform arithmetic with such large numbers

(even using the Chinese Remainder Theorem). So a good technique would be to combine this technique with a Monte Carlo technique, so that in the non-zero case a result would with high probability be found relatively quickly. It would still be necessary to perform the large number evaluations to prove an exact zero. We first may use the *Landau-Mignotte* bound to calculate an upper bound on the possible size of any coefficients of the factors of the polynomial, which we shall denote $p(x)$. This bound we shall denote B.

Theorem 4.4 (Univariate zero test, large value method)
given $p(x) \in \mathbf{Z}[x]$
If $p(b) = 0$ *where* $b > B \wedge b \in \mathbf{Z}$ *then* $p(x) = 0$
Proof :
$p(b) = 0 \Rightarrow (x - b)|p(x) \vee p(x) = 0$
the second option is the only possibility as b is too large

□

Definition 4.1 *let multibound*$(f(x_1, \cdots, x_n))$ *be defined as the point* $(B_1, \cdots, B_n)$ *where* B_i *is one plus the bound on the right hand side of the inequality 4.4 applied to the polynomial* $f(B_1, \cdots, B_{i-1}, x_i, \cdots, x_n)$.

Throughout the following theorem we shall define d_{x_i} to be $\deg_{x_i}(f(x_1, \cdots, x_n))$ in $f(x_1, \cdots, x_n)$. Also Σ shall be used to denote $d_1 + \cdots + d_n$ and b_i to denote a bound to the coefficients of the polynomial $f(B_1, \cdots, B_{i-1}, x_i, \cdots, x_n)$.

Theorem 4.5 (Multivariate zero test, large value method)
If $f(B_1, \cdots, B_n) = 0$ *where* $(B_1, \cdots, B_n)$ *equals multibound*$(f(x_1, \cdots, x_n))$, *then* $f(x_1, \cdots, x_n) = 0$.
Proof :
Base for Induction:
Since $f(B_1, \cdots, B_n) = 0$ *then either* $(x_n - B_n)$ *is a factor of* $f(B_1, \cdots, B_{n-1}, x_n)$ *or* $f(B_1, \cdots, B_{n-1}, x_n)$ *is identically zero. However* B_n *is chosen to large such that the former is true.*
$\therefore\ f(B_1, \cdots, B_{n-1}, x_n) = 0$
Inductive case:
as inductive hypothesis assume that $f(B_1, \cdots, B_i, x_{i+1}, \cdots, x_n) = 0$. *Now either*

$(x_i - B_i)$ *is a factor of* $f(B_1, \cdots, B_{i-1}, x_i, \cdots, x_n)$ *or* $f(B_1, \cdots, B_{i-1}, x_i, \cdots, x_n)$ *is identically zero. However* B_i *is chosen to large for the former to be true.*
$\therefore f(B_1, \cdots, B_{i-1}, x_i, \cdots, x_n) = 0$
$\therefore$ *by induction we have* $f(x_1, \cdots, x_n) = 0$

□

4.13.1 Complexity Analysis

We look at the complexity analysis for the size of the evaluation of $f(B_1, \cdots, B_n)$
$f(B_1, \cdots, B_n) = O((\cdots((B_n^{d_n})B_{n-1})^{d_{n-1}} \cdots B_1)^{d_1})$
where:
$B_1 = 2^{\Sigma}||f(x_1, \cdots, x_n)|| < 2^{\Sigma}b_1$,
$B_2 = 2^{\Sigma}b_2 < 2^{\Sigma}d_2(2^{\Sigma}b_1)^{d_1}$,
and $B_i = 2^{\Sigma}d_i(\cdots(2^{\Sigma}d_2(2^{\Sigma}b_1)^{d_1})^{d_2}\cdots)^{d_{i-1}}$
We notice that the number of modular fields needed to perform this evaluation is doubly exponential.

4.14 Monte-Carlo equality checking

Due to the high time complexity overheads of deterministic equality checking, we shall consider a Monte Carlo equality checking method. We take this method from Schwartz [10]. The basic idea is to evaluate the SLP at a number of random points in some space. If the space is big enough then we are guaranteed that at some points the SLP will not evaluate to zero, unless the polynomial which is represented by the SLP is identically zero. By making the space of a sufficient size we may ensure that the probability of an incorrect answer to the solution of the question "is the SLP identically zero or not?" is less than a given value. We quote two results due to Schwartz:

Theorem 4.6 *Suppose that* Q *is a polynomial in the variables* $x_1, \cdots, x_n$ *and that* Q *is not identically zero, suppose also that* $I_1, \cdots, I_n$ *are any non-empty sets of elements in the domain of the coefficients of* Q. *Let the degree of* Q *in the*

variable x_i be denoted by d_i. Then in the set $I_1 \times \cdots \times I_n$, Q has at most :

$$\begin{aligned} & d_1|I_2|\cdots|I_n| + d_2|I_1||I_3|\cdots|I_n| + \cdots + d_n|I_1|\cdots|I_{n-1}| \\ = \; & |I_1 \times \cdots \times I_n| \sum_{i=1}^{n} \frac{d_i}{|I_i|} \end{aligned} \tag{4.6}$$

zero points.

□

Corollary 4.1 *Let $I = I_1 = \cdots = I_n$ and let $|I| \geq c\deg(Q)$ then if Q is not identically zero, the number of elements of $I_1 \times \cdots \times I_n$ which are zeros of Q is at most $c^{-1}|I|^n$.*

□

We can therefore test a purported identity $Q = 0$ by the following Monte Carlo procedure. Choose I such that $|I| \geq C \deg(Q)$ with C significantly greater than 1. Then in order to test the purported identity where the probability of failure is bounded by ϵ we must choose an N, such that $\epsilon > C^{-N}$. Now we randomly choose N points $y = (y_1, \cdots, y_n)$ from $I_1 \times \cdots \times I_n$. If $Q(y)$ is zero for every y then we have proven the above identity with probability of failure less than ϵ.

One of the more efficient ways of evaluating SLPs is by using modular arithmetic since multiplication and division become much cheaper operations. In order to investigate how we may use modular arithmetic to perform Monte Carlo equality tests we make the following definitions. These and the following results are also taken from Schwartz.

Definition 4.2 *Let $Q \in Z[x_1, \cdots, x_n]$*
(a) A modular zero of Q is an $(n+1)$-tuple $(i_1, \cdots, i_n, p)$ of integers, the last integer p being prime, such that $Q(i_1, \cdots, i_n) = 0 \bmod p$.
(b) We write maxv(Q,k) for the maximum absolute value which Q can assume on the rectangle $|x_j| \leq k, j = 1, \cdots, n$.

Theorem 4.7 *Suppose that the suppositions of theorem 4.6 are satisfied. Suppose also that Q is over the integers, that $I = I_1 = \cdots = I_n = \{i | 0 \leq i \leq k\}$*

and that $L = maxv(Q,k)$. *Let* J *be any finite set of primes, and suppose that the product of any* m *of the primes in* J *exceeds* L. *Then in the set* $I_1 \times \cdots \times I_n \times J$, Q *has at most:*

$$|I_1 \times \cdots \times I_n \times J|(\sum_{i=1}^{n} \frac{d_i}{|I_i|} + \frac{m}{|J|}) \tag{4.7}$$

modular zero points.

□

Corollary 4.2 *Let* $2k+1 \geq c\deg(Q)$ *and suppose that the product of the* $c^{-1}|J|+1$ *smallest primes in* J *exceeds* $maxv(Q,k)$. *Then if* Q *is not identically zero, the number of elements of* $I_1 \times \cdots \times I_n \times J$ *which are zeros of* Q *is at most* $2c^{-1}|I|^n m$

In much the same way as for corollary 4.1 we may test a purported identity $Q = 0$ by the following Monte Carlo procedure. Set $2k + 1 \geq c\deg(Q)$, select J to be the smallest set s.t. $\prod_{j \in J} j > maxv(Q,k)$ and choose N random points s.t. $\epsilon < c^{-N}$, where ϵ is the probability of failure. If we get modular zeros at each of these points, then we have our purported identity with the required probability of failure.

Chapter 5

Interpolation techniques

By interpolation of a polynomial we mean a technique for determining the coefficients of the polynomial. The first couple of techniques we consider require the evaluation of the polynomial at a number of points.

5.1 The General solution to the interpolation problem

We show how to perform interpolation of the polynomial $p(x_1, \cdots, x_m)$ given by the following equation, with respect to the variable x_1:

$$p(x_1, \cdots, x_m) = c_0(x_2, \cdots, x_m)x_1^0 + c_1(x_2, \cdots, x_m)x_1^1 + \cdots + c_n(x_2, \cdots, x_m)x_1^n \tag{5.1}$$

That is, to determine the coefficients $\{c_i(x_2, \cdots, x_m) \ : 0 \leq i \leq n\}$, which may themselves be multivariate polynomials.

We take $n+1$ arbitrary points, for example, $\xi_0 = 0, \cdots, \xi_n = n$. After performing the evaluation of p at $x = \xi_0$ to $x = \xi_n$, we have the following set of equations:

$$
\begin{aligned}
&c_0 + c_1\xi_0 + \cdots + c_n\xi_0^n = p(\xi_0)\\
&c_0 + c_1\xi_1 + \cdots + c_n\xi_1^n = p(\xi_1)\\
&\ddots \quad \ddots \quad \ddots \quad \ddots\\
&c_0 + c_1\xi_n + \cdots + c_n\xi_n^n = p(\xi_n)
\end{aligned}
\tag{5.2}
$$

This is equivalent to the Vandermonde matrix equation:

$$
\begin{pmatrix}
1\,, & \xi_0^1\,, & \cdots & ,\xi_0^n\\
1\,, & \xi_1^1\,, & \cdots & ,\xi_1^n\\
\ddots & \ddots & \ddots & \ddots\\
1, & \xi_n^1, & \cdots & ,\xi_n^n
\end{pmatrix}
\begin{pmatrix} c_0\\ c_1\\ \vdots\\ c_n \end{pmatrix}
=
\begin{pmatrix} p(\xi_0)\\ p(\xi_1)\\ \vdots\\ p(\xi_n) \end{pmatrix}
$$

we shall denote these matrices

$$VC = P$$

In this notation we see that the interpolation problem is equivalent to inverting V to get V^{-1}. Then $C = V^{-1}P$ which is the answer we require.

5.2 Lagrange Interpolation

Following Zippel [30] we make the following deductions.
Multiply V by a generic inverse to get the following:

$$VV^{-1} = 1$$

$$
\begin{pmatrix}
1\,, & \xi_0^1\,, & \cdots & ,\xi_0^n\\
1\,, & \xi_1^1\,, & \cdots & ,\xi_1^n\\
\ddots & \ddots & \ddots & \ddots\\
1, & \xi_n^1, & \cdots & ,\xi_n^n
\end{pmatrix}
\begin{pmatrix}
p_{00}, & p_{01}\,, & \cdots & ,p_{0n}\\
p_{10}\,, & p_{11}\,, & \cdots & ,p_{1n}\\
\ddots & \ddots & \ddots & \ddots\\
p_{n0}, & p_{n1}, & \cdots & ,p_{nn}
\end{pmatrix}
= 1
\tag{5.3}
$$

We see that the j^{th} element of the i^{th} row of the LHS of this product is:

$$p_{0j} + p_{1j}\xi_i + \cdots + p_{nj}\xi_i^n = P_j(\xi_i) \tag{5.4}$$

where $P_j(x)$ is a polynomial of degree n in x, with the matrix:

$$\begin{pmatrix} P_0(\xi_0) & \cdots & P_n(\xi_0) \\ \vdots & \ddots & \vdots \\ P_0(\xi_n) & \cdots & P_n(\xi_n) \end{pmatrix}$$ equal to the identity.

since $P_j(\xi_i)$ is zero if $i \neq j$ and one when $i = j$, we may deduce the following formula:

$$P_j(x) = \prod_{\substack{i \neq j \\ 0 \leq i \leq n}} \frac{x - \xi_i}{\xi_j - \xi_i} \tag{5.5}$$

The numerator ensures that all the off diagonal entries are zero (as there will be one $(\xi_i - \xi_i) = 0$ factor), whilst the denominator ensures that the diagonal terms are one (the $\frac{\xi_i - \xi_i}{\xi_i - \xi_i}$ term is omitted, every other factor of $\frac{\xi_j - \xi_i}{\xi_j - \xi_i} = 1$).
Returning to the problem of finding the c_i in equation 5.2 , which may now be writen as:

$$\begin{pmatrix} c_0 \\ \vdots \\ c_n \end{pmatrix} = \begin{pmatrix} p_{00} & \cdots & p_{0n} \\ \vdots & \ddots & \vdots \\ p_{n0} & \cdots & p_{nn} \end{pmatrix} \begin{pmatrix} p(\xi_0) \\ \vdots \\ p(\xi_n) \end{pmatrix} \tag{5.6}$$

We may now form the set of equations:

$$c_i = p_{i0}p(\xi_0) + \cdots + p_{in}p(\xi_n) \tag{5.7}$$

or:

$$c_i = \text{coef}(P_0(x), x^i)p(\xi_0) + \cdots + \text{coef}(P_n(x), x^i)p(\xi_n) \tag{5.8}$$

substituting these expressions into the formulae 5.1 we get:

$$\begin{aligned} p(x) = \; & \text{coef}(P_0(x), x^0)p(\xi_0) + \cdots + \text{coef}(P_n(x), x^0)p(\xi_n) \\ & + (\text{coef}(P_0(x), x^1)p(\xi_0) + \cdots + \text{coef}(P_n(x), x^1)p(\xi_n))x \\ & + \cdots \\ & + (\text{coef}(P_0(x), x^n)p(\xi_0) + \cdots + \text{coef}(P_n(x), x^n)p(\xi_n))x^n \end{aligned} \tag{5.9}$$

There is no point in extracting the coefficients of x^i in this equation, we may write it more succinctly as

$$p(x) = P_0(x)p(\xi^0) + \cdots + P_n(x)p(\xi^n) \tag{5.10}$$

or

$$p(x) = \sum_{j=0}^{n} \left(\prod_{\substack{i=0 \\ i \neq j}}^{n} \frac{(x - \xi_i)}{(\xi_j - \xi_i)} \right) p(\xi_j) \tag{5.11}$$

5.2.1 Mixed Representation

So how does this allow us to determine the coefficients? if we simply produce an SLP equivalent to p we are no wiser than before, we simply have a bigger SLP for p, with x_1 as the main variable. What we must do is implement a mixed representation version of the sparse arithmetic functions:

addition, subtraction, multiplication and division by constants

and then apply this to equation 5.2.

The representation used for the polynomials will be mixed, in that we shall store vectors of SLPs, where the SLPs represent the coefficent polynomials. The coefficient of x_1^i taking up position $i+1$ in the vector.

We shall look at how to implement these operations:

- Summation and Subtraction:
  In the following assume WLOG. that $n > m$,

$$(c_0, c_1, \cdots, c_n) \pm (d_0, d_1, \cdots, d_m) = (c_0 \pm' d_0, c_1 \pm' d_1, \cdots, c_m \pm' d_m, \cdots, c_n)$$

- Multiplication:

$$(c_0, c_1, \cdots, c_n) * (d_0, d_1, \cdots, d_m) = (b_0, \cdots, b_{m+n}) \text{ where } b_k = \sum_{i+j=k}{}' c_i *' d_j$$

- Division by constants:
  If we may calculate inverses in our base ring, then the problem reduces to calculating the inverse then creating a multiplication node, otherwise we are forced to include a division node in the SLP, so removing the division free attribute of the SLP.

In the above $+'$, $-'$ and $*'$ denote the lazy SLP operations, that is the creation of an SLP node.

5.2.2 Formation of $P(z)$

An efficient way to calculate $P(z)$ involves the initial calculation of the polynomial $\prod_{i=0}^{n}(z - x_i)$. Then we require to divide this by $(z - x_i)$ for each $0 \leq i \leq n$ however as we are using a lazy evaluation technique, we find it easier to avoid the multiplication in the first place. We borrow a technique from lattice theory to produce the set of products $\{\prod_{\substack{i=0 \\ i \neq j}}^{n}(z - x_i) | 0 \leq j \leq n\}$, whilst making as much reuse of values as possible.

In the following * represents multiplication nodes of an SLP. The two lines leading into the * (from above) show where information is flowing from and where information is flowing to, that is from the node at the top of the line, to the node at the bottom of the line. When we refer to nodes higher up the tree, the direction is in the direction of the flow of information, i.e.. "higher up" means lower down the diagram.

The first step is to produce a binary multiplication tree, for calculating the product $\prod_{i=0}^{n}(z - x_i)$ except that the top node will be removed, ie.

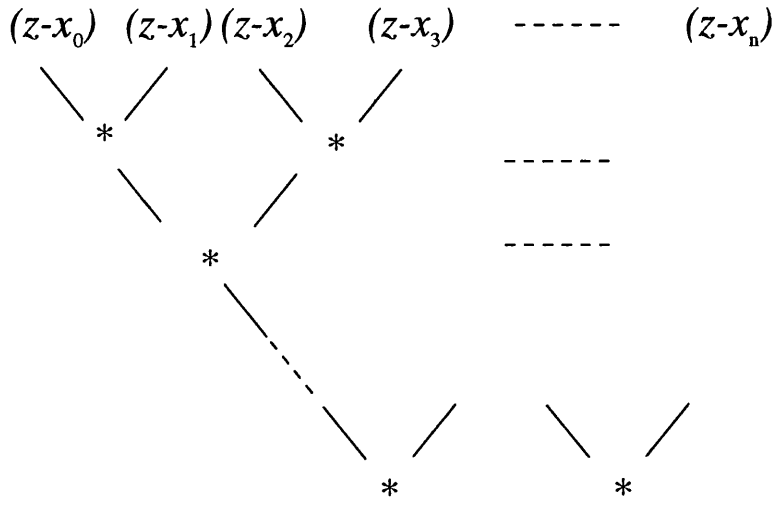


Figure 5.1: The initial half of the product network

The next step is to complete the product via multiplication by nodes as high up the network as possible, we find a surprisingly neat way to do this.

e.g. consider the case $n = 3$, in the following we shall label the factor $(z - x_i)$ by the number i,

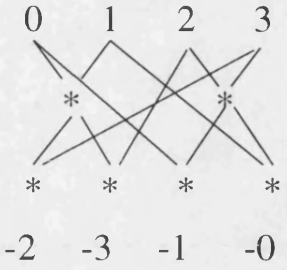


Figure 5.2: An example network where the final polynomial has degree three

The negative numbers at the top of the network denote products which are missing the factor indicated.
We shall now look at a more complex case, that where $n = 7$. The first half of the tree looks like:

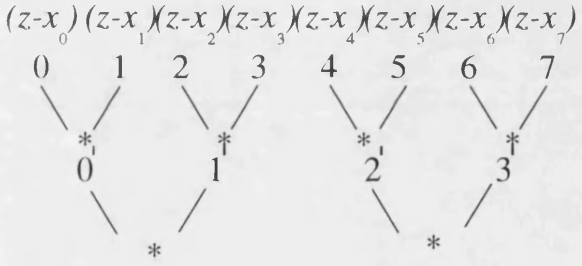


Figure 5.3: The first half of an eight-network

Now consider the first stage of the second half of the tree:

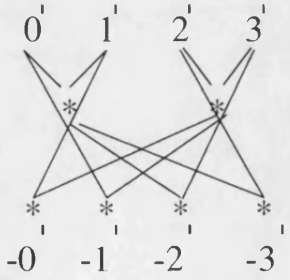


Figure 5.4: The first stage of the second half of an eight-network

A network constructed in such a way, which has four inputs and four outputs, we

shall call a four-network.
Now we must complete the eight-network, we do this in the following way:

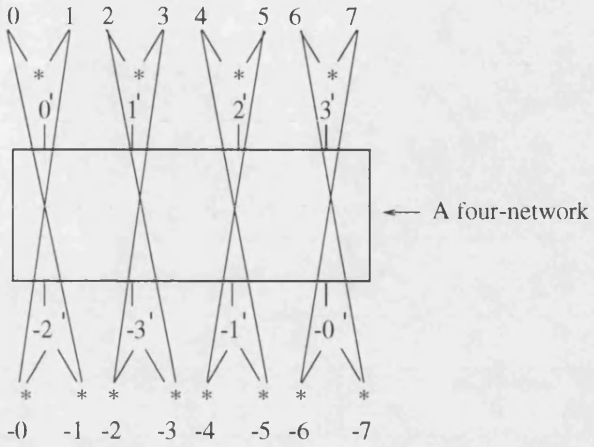


Figure 5.5: An eight-network

We shall now consider how to generalise this to that of an n-network, where $n = 2^m$ for $m \in \mathbf{Z}$.

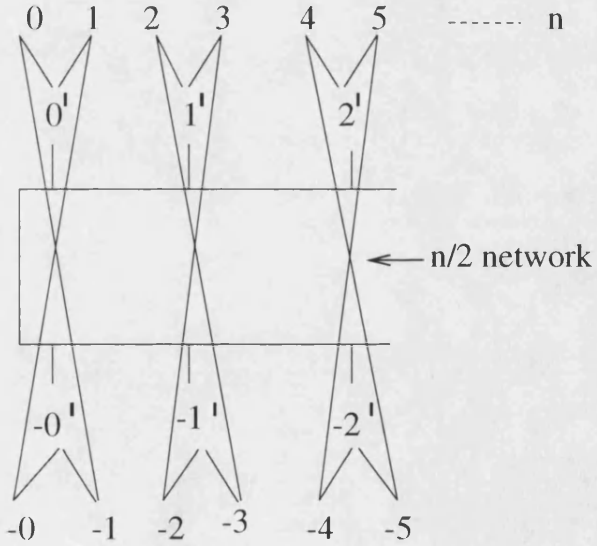


Figure 5.6: A diagram describing a network with $n = 2^m$ inputs

These products having been constructed, we multiply by the evaluated SLPs $p(x_i)$ then do the divisions as described earlier. We use a binary tree for constructing

the summations, since in general a binary tree will be the network with smallest depth which represents the summation.

5.2.3 Complexity analysis

In order to perform the complexity analysis for our Lagrange interpolation, it is useful to split the calculation into several stages:

- The n evaluations:
  $length = O(n * l_{eval})$ $\quad$ $depth = O(d_{eval})$
  where d_{eval} is the maximum depth of an evaluation and
  l_{eval} is the maximum length of an evaluation.

- Construction of the nodes which will be the inputs to the multiplication network, viz:
  $$(x - x_0), \cdots, (x - x_n)$$
  due to reuse of nodes, we must only store one input node of 1, which is the coefficient of x in each factor. And $n + 1$ input nodes, $x_0, \cdots x_n$ which are the evaluation points. That is a total of $n + 2$ nodes.

- The construction of the multiplication network:
  This network may be split into two sections,
  The first section:
  In this section of the network, the network is as close to a binary tree as we can get with respect to the polynomial multiplications. There are $O(\lceil log_2 n \rceil)$ levels in this binary tree, were the numbering starts from one (from now on in this analysis we shall denote $\lceil log_2 n \rceil$ by λ). For the $(\lambda - i + 1)^{th}$ level, there are $O(2^i)$ polynomial multiplications per level.
  For multiplication of two polynomials of degree d, we shall consider the cost: the length costs $O(d^2)$ multiplications and $O(d^2)$ additions.
  The i^{th} level involves multiplications of pairs of polynomials each with degree 2^{i-1} In summary we may say that the top section will contain:
  $O(\sum_{i=2}^{\lambda}(2^{i-1})^2)$ multiplication nodes and $O(\sum_{i=2}^{\lambda}(2^{i-1})^2)$ addition nodes.
  The second section:

In this section of the network, there are the same number of polynomial multiplications, however the arguments come from different levels of the tree, one argument is taken from the preceding level, one argument will be taken from the top section of the tree, in the following manner: on the $(\lambda + i)^{th}$ level which is in the bottom section, we shall take one argument from the $(\lambda - i)^{th}$ level which is in the top section.
For two polynomials p, q with degrees d_p, d_q respectively, a polynomial multiplication will take $d_p d_q$ base ring multiplications and $d_p d_q$ base ring additions.
For the i^{th} level, the arguments have degrees: $O(2^{\lambda-i})$ and $O(2^{\lambda}-2^{\lambda-i+1}-1)$ respectively. In summary we may say that the bottom section will contain: $O(\sum_{i=2}^{\lambda} 2^{\lambda-i}(2^{\lambda}-2^{\lambda-i+1}-1))$ multiplication nodes and $O(\sum_{i=2}^{\lambda} 2^{\lambda-i}(2^{\lambda}-2^{\lambda-i+1}-1))$ addition nodes.

- Division by the valuation point differences:
  In this section of the network there will be $n(n-1)$ division nodes (division of n polynomials, each with $n-1$ coefficients).

- Construction of the final summation tree: In this section of the tree there will be $O(n^2)$ addition nodes.

- So the final complexity of the size of the SLP is:

$$\begin{aligned} S &= O(2\sum_{i=2}^{\lambda}(2^{i-1})^2 + 2\sum_{i=2}^{\lambda} 2^{\lambda-1}(2^{\lambda} - 2^{\lambda-i+1} - 1) \\ &\quad + nl_{eval} + n + 2 + n(n-1) + n^2) \qquad (5.12) \\ &= O(2\sum_{i=2}^{\lambda}((2^{i-1})^2 + 2^{\lambda-1}(2^{\lambda} - 2^{\lambda-i+1} - 1)) \\ &\quad + 2 + nl_{eval} + 2n) \qquad (5.13) \end{aligned}$$

now $\sum_{i=2}^{\lambda}(2^{i-1})^2 = O(\frac{2^{\lambda}}{2})^2 = O(n^2)$
$\therefore$

$$\begin{aligned} S &= O(n(l_{eval} + 2n) + 1 + n^2 + \frac{n}{2}(n - \frac{2n}{2^i} - 1)) \qquad (5.14) \\ &= O(n(l_{eval} + n)) \qquad (5.15) \end{aligned}$$

If the number of variables is significant the value $O(nl_{eval})$ contribution

will be closer to l_{eval} since the coefficients with respect to the interpolation variable will be independant of that variable and therefore the separate evaluations will have more shared structure. In this case we see:

$$S = O(n^2)$$

5.3 FFT Interpolation

The interpolation method described in this section is based on a method due to Zippel ([30], chapter 13. section 4.), the FFT Interpolation method (or Fast Fourier Transform Interpolation method) uses evaluations of the polynomial at special points which have nice symmetry characteristics, viz. roots of unity, see section 5.3.1. This allows us to manipulate the Vandermonde matrix so that it is sparse.The previous interpolation algorithm took $O(n^2)$ operations. We find that we may factorise the Vandermonde matrix into $\log n$ matrices, each of which has only 2 non-zero elements in each row. Then multiplying by one factor takes $O(n)$ operations and multiplying by all of them takes $O(n \log n)$ operations.

Notation
If ξ is a primitive n^{th} root of unity (i.e.$\xi^n = 1 \ \wedge \ 0 \leq m < n \Rightarrow \xi^m \neq 1$).

We denote the $n \times n$ Vandermonde matrix, $\begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & \xi & \cdots & \xi^{n-1} \\ 1 & \xi^2 & \cdots & \xi^{2(n-1)} \\ \vdots & & \ddots & \vdots \\ 1 & \xi^{n-1} & \cdots & \xi^{(n-1)^2} \end{pmatrix}$ by $F_n(\xi)$.

The diagonal matrix with diagonal elements $\{1, \xi, \xi^2, \cdots, \xi^{n-1}\}$ is denoted by Ω_n. The identity element with size n is denoted by I_n. We need the following theorem:

Theorem 1 *Fourier Inverse transformation*
If ξ is an n^{th} root of unity. then,

$$F_n(\xi) \cdot F_n(\xi^{-1}) = nI_n$$

Sketch Proof: The left hand side of the given identity is clearly the product of two matrices:
The diagonal elements of this product are clearly all n.
For the theorem to hold we require that every other element is zero.
Consider the distinct n^{th} primitive root of unity ξ_n, by definition the following equation holds:

$$\begin{aligned} z^n - 1 &= (z - \xi_n^0)(z - \xi_n)(z - \xi_n^2) \cdots (z - \xi_n^{n-1}) && (5.16) \\ &= z^n - \left(\sum_{k=0}^{n-1} \right) z^{n-1} + \cdots + (-1)^n \prod_{k=0}^{n-1} \xi_n^k && (5.17) \end{aligned}$$

It may be seen by comparing coefficients that all the required sums of roots of unity are zero.

□

The FFT Interpolation strategy is based on the following observation:

Let $n = 2m$ be an even integer. Define Π_n to be a permutation matrix such that the first m columns of $A\Pi_n$ are the even columns of A and the second m columns are the odd columns of A. Then:

$$F_n \Pi_n = \begin{pmatrix} I_m & \Omega_m \\ I_m & \Omega_m \end{pmatrix} \begin{pmatrix} F_m & 0 \\ 0 & F_m \end{pmatrix} = \begin{pmatrix} F_m & \Omega_m F_m \\ F_m & \Omega_m F_m \end{pmatrix} \qquad (5.18)$$

□

Comparing the cost of computing the Fourier transform of an n-vector by multiplying by F_n and by using the factorization on the right hand side of 5.18, the former takes n^2 multiplications, whilst the latter requires only $n^2/2$ multiplications. Continuing in this manner we may get a factorization of $F_n \Pi_n \Pi'_{n/2}$, this may be repeated for $\log n$ times, rendering the matrix completely diagonal.
This algorithm appears to be recursive, however it transpires that performing all the permutations at once is simpler. The result is a single reordering of the columns of the original matrix which may be determined from the *bit reversal* of the indices of the columns. By bit reversal we mean take the binary representation of the index, where the columns are indexed from 0, then reverse the bits to give a new number. This will give us the new order for the columns. We may perform this algorithm using the following recursive algorithm.

binary_reversal
input : binary rep

1. if binary rep < 2 then return binary rep
2. even $\leftarrow$ the even bits of binary rep; odd $\leftarrow$ the odd bits of binary rep
3. return append(binary_reversal even,binary_reversal odd)

5.3.1 Modular Roots of Unity

By an n^{th} root of unity, we mean a root of the polynomial $x^n - 1$. Clearly not all fields hold a full quota of n^{th} roots of unity. For $n = 1$; 1 is clearly the only root of $x^1 - 1$. For $n = 2$, $x = 1$ and $x = -1$ where -1 is the additive inverse of 1, are the two quadratic roots of $x^2 - 1$. However an obvious example shows that this doesn't apply when $n > 2$. Consider the four quartic ($n = 4$) roots of unity, that is $\{1, -1, i, -i\}$ (we denote the square root of -1 as i), these are not members of the Real numbers, however they are members of the Gaussian Rationals (the rational numbers extended by i). For other values of n we must make further field extensions to represent all the roots of unity exactly.
We could follow an alternative route, that is to perform arithmetic in special modular fields which contain n, n^{th} roots of unity. These modular fields have prime moduli, where the primes are special primes called Fourier primes. Fourier primes are of the form $p_f = (n = 2^m)k + 1$. We must find a *primitive root* of unity, that is $\xi \in \mathbf{Z}_{p_f}$ s.t. $\xi^n = 1$ and $\xi^r \neq \xi^l$ for $r \neq l$ and $0 \leq r, l < n$. We may then calculate all the other roots of unity by multiplying ξ by itself n times. We have an algorithmic method for calculating ξ, taken from Lipson (Algorithm 1, chapter IX) [15].

5.3.2 Method for calculating modular roots

We take much of this section from Lipson [15]. For our first result we shall deduce the form that the prime modulus must take:

In the following theorem we shall denote the multiplicative group of $\mathbf{Z}_p = \mathbf{Z}_p/\{p\}$ under the operation of multiplication by $\mathbf{Z}_p^*$.

Theorem 2 *$\mathbf{Z}_p$ has a primitive N^{th} root of unity if and only if $N|(p-1)$.*

Proof. By Lagrange's Theorem, the order of a group element divides the order of the group. Since $\mathbf{Z}_p^*$ has order $p-1$ we obtain the 'only if' direction.
Now $\mathbf{Z}_p$ has a primitive element by 'Fermat's little theorem', call it α. We see that $\alpha^{(p-1)/N}$ has order N in $\mathbf{Z}_p^*$, making $\alpha^{(p-1)/N}$ the primitive N^{th} root of unity. □

So in order to find primes p s.t. the modular field has the requisite number of roots, we must find p s.t. $2^m|(p-1)$, i.e. primes of the form $p = 2^m k + 1$, where k is some positive integral constant. These are the Fourier Primes mentioned earlier.
Now that we have found a field containing the requisite number of roots, it is necessary to find the primitive N^{th} root of unity. The following theorem appertains to that.

Theorem 3 *Let $\alpha \in \mathbf{Z}_p$. α is primitive if and only if $\alpha^{(p-1)/q} \neq 1$ in $\mathbf{Z}_p$ for any prime factor q of $(p-1)$.*

Proof. It is trivial to show the if direction.
For the only if direction:
assume that α has order $n < p-1$. Then n divides $p-1$ by Lagrange's Theorem, so that $p-1 = kn$. Let $k = qr$, where q is a prime in the prime factorization of k. But then $p-1 = qrn$, so that q is also a prime in $p-1$'s (unique) prime factorization. We then have,

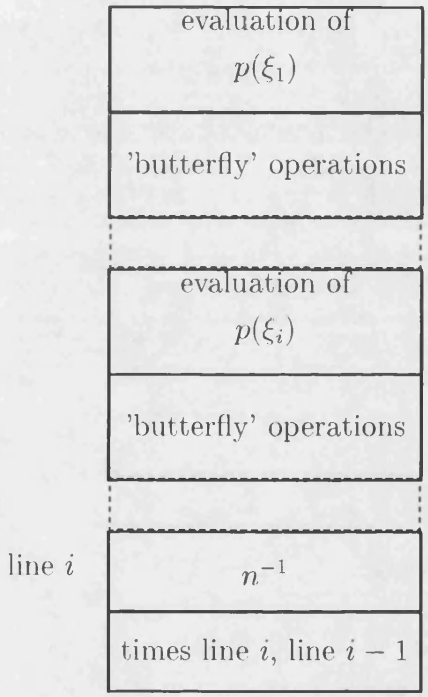
$$\alpha^{(p-1)/q} = \alpha^{rn} = (\alpha^n)^r = 1$$

□

As we will be working on 32 bit computers, it would be sensible to start with the biggest (Fourier prime) modulus less than 2^{32}, then the next one down and so on, until we reached the smallest (fourier prime) modulus greater than one. In the unlikely event that this does not supply enough primes, we would start just below 2^{64} working down to just above 2^{32} and so on. This way modular arithmetic could be done most efficiently.

5.4 Optimisations

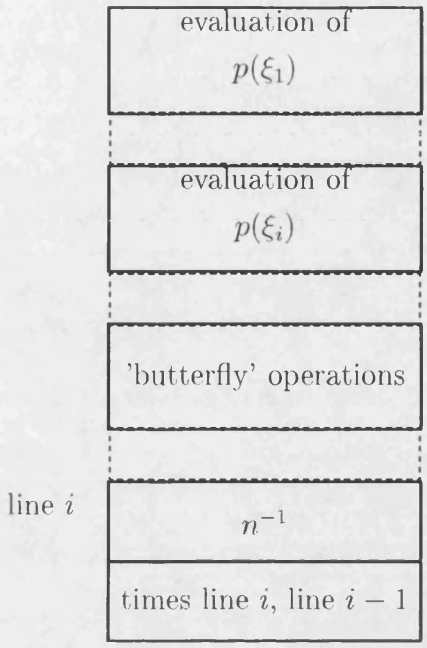
Our original implementation of the FFT interpolation algorithm consisted of taking the evaluation of the SLP at each of the modular roots of unity, then treating each of these as individual SLPs, it was then necessary to perform arithmetic on them. This resulted in much unnecessary merging of large objects. We have no accurate timings for this naïve implementation, however estimated timings for the interpolation of a bi-variate polynomial, with degree 15 in the interpolation variable and degree 20 in the other variable, gave a result in approximately 4 hours. The coefficients returned are in the following form:



5.4.1 Collection of evaluations

The first optimisation performed is the following, we initially perform evaluation of the polynomial at each of the roots of unity (ξ_i) resulting in n SLPs, each of these are merged together using the algorithm BetterMerge$_n$ of section 4.4.4, the butterfly operations may now be performed, this saves a lot of program merging as we simply append the new operations. The resulting program will be of the

following form:



timings of the test problem already detailed using this implementation were approximately two hours.

5.4.2 Complexity Analysis

The SLP resulting from the FFT interpolation will be made up from two parts:

- The n partial evaluations, this section will be $O(nl)$ in length, where l is the length of the original SLP,
- The butterfly operations, this section will be $\sum_{i=0}^{\log n} 4\left(\frac{n}{2^i}+1\right)$ in length, as there will be $\log n$ steps, and for the i^{th} step we must make $\frac{n}{2^i}+1$ calls to the basic fft operation which involves two multiplications and two divisions.

so in summary the FFT interpolation produces an SLP which has length:

$$O(nl + \sum_{i=0}^{\log n} 4\left(\frac{n}{2^i}+1\right))$$

The previous two methods are essentially *Black Box* methods of interpolation. In essence it is not neccessary to know the representation in order to perform the interpolation so long as we know how to evaluate the objects and also to be able to construct new ones. The next method makes specific use of knowledge about the SLP representation. The previous techniques require evaluation at $d+1$ points, where d is the degree of the polynomial object to be interpolated. The following method requires no evaluations, except in the case of non-division free SLPs in which case the evaluation is only necessary for dealing with division nodes.

5.5 Direct translation from polynomial program to coefficient programs

A further method of interpolating SLPs has been suggested by Kaltofen [11]. We first look at his method in the case of division free SLPs.
The basic idea is to look at each node in turn starting at the base of the SLP, then to determine what effect each node will have on each coefficient with respect to the interpolation variable:
If the program represents a polynomial having degree $< d$, we loop through the program. For the instruction which we shall label v_λ, we construct a program as follows.
The SLP which has as its return node v_λ may be thought of as a polynomial in its own right. We intend to construct a program to calculate $d+1$ of its coefficients, for each node v_λ in the original program, we shall label pointers to the new coefficient instructions $w_{\lambda,\delta}$ for $\delta \in \{0 \cdots d\}$. That is, the new instruction pointed to by $w_{\lambda,\delta}$ corresponds to the $(\delta+1)$th coefficient (the coefficient of x_1^δ) of the polynomial which corresponds to the program which has return node this v_λ. Denote the return program as *prog*.

- start *prog* with the two constant nodes: 0,1.
- initialise w so that every element points to the 0 node, i.e. is 1.
- If v_λ is a constant node then add v_λ to *prog* and set $w_{\lambda,0}$ to the position of v_λ in *prog*.

- If v_λ is an input node then we have two cases:
  - either v_λ is equal to the interpolation variable, in which case we set $w_{\lambda,1}$ to point to 1, i.e. line 2.
  - otherwise we set $w_{\lambda,0}$ to point to v_λ and add v_λ to prog.
- If v_λ is an operation node, we may write v_λ as $v'_\lambda \circ_\lambda v''_\lambda$, denote the elements of w corresponding to v'_λ, v''_λ as $w'_{\lambda,i}$, $w''_{\lambda,i}$ respectively for the index i. now we have two cases:
  - the case that $\circ_\lambda$ is plus or minus:
    for $\delta \in \{0 \cdots d\}$ append the operation node $\circ_\lambda$: $w'_{\lambda,\delta}$, $w''_{\lambda,\delta}$ to *prog* and set $w_{\lambda,\delta}$ to point to this instruction.
  - the case that $\circ_\lambda$ is times:
    for $\delta \in \{0 \cdots d\}$
    append instructions to compute the summation $\sum_{\substack{i+j=\delta \\ 0 \le i,j \le d}} w'_{\lambda,i} w''_{\lambda,j}$, we make this a binary tree, to limit the depth of the SLP. Then set $w_{\lambda,\delta}$ to point to the final element in this set of instructions.
- The return instructions for the coefficients will be the instructions $w_{\lambda_f,\delta}$ for $\delta \in \{0, \cdots, d\}$ where v_{λ_f} is the return instruction for the input program.

Example

We perform the direct interpolation of an SLP which represents the polynomial which in sparse form is $xy + 3$, viz:

```
          Input Program
line 1 :  Input node      : x
line 2 :  Input node      : y
line 3 :  Constant node   : 3
line 4 :  Operation node  : (times line 1 , line 2)
line 5 :  Operation node  : (plus line 4 , line 3)   >
```

Initial program is:

line 1 : Constant node : 0
line 2 : Constant node : 1

We shall look at each line of Input Program in turn

line number	w_i	New Instructions
1	[1, 2]	[ ]
2	[3, 1]	[Input node : y]
3	[4, 1]	[Constant node : 3]
4	[5, 6]	[Operation node : (times 1,3)
		Operation node : (times 2,3)
		Operation node : (times 1,1)
		Operation node : (plus 6,7)]
5	[9, 10]	[Operation node : (plus 4,5) $>_0$
		Operation node : (plus 1,8) $>_1$]

The node indicated by $>_i$ represents the coefficient of x^i.
If we include the optimisations of section 5.6 we get:

line 1 : Constant node : 0
line 2 : Constant node : 1
line 3 : Input node : y $>_1$
line 4 : Constant node : 3 $>_0$

5.5.1 Treatment of division nodes

We shall now briefly describe the method used to deal with non-division free SLPs.
If v_λ is a division node, then we may write v_λ as $v'_\lambda \circ_\lambda v''_\lambda$ where $\circ_\lambda$ is the division operator. We essentially find an expression for the *Newton approximation* of $1/v''_\lambda$, we can then use the algorithm for the multiplication nodes described above to find the expression for the complete node.
We make the approximation mod x_1^m where m is greater than d. Now even though some of the intermediate division nodes may evaluate to rational functions, we

know that the output node is purely polynomial, and so these approximations will give an exact final answer.

In the following algorithm description $g(x_1, \cdots, x_n)$ represents the function corresponding to the SLP node v''_λ, whilst w_δ represent the functions corresponding to the coefficient of x_1^δ in $g(x_1, \cdots, x_n)$.
The algorithm follows:

- $\alpha_0 \leftarrow 1/g(0, x_2, \cdots, x_n)$
  FOR $i \leftarrow 1, \cdots, \lceil \log(d) + 1 \rceil$ DO
    At this point α_{i-1} is the $2^{i-1}-1$st order approximation of $g(x_1, \cdots, x_n)$
    $\alpha_i \leftarrow 2\alpha_{i-1} - \alpha_{i-1}^2(\sum_{\delta=0}^{d} w_\delta x_1^\delta) \bmod x_1^{\min(d+1, 2^i)}$

We shall discuss the Newton method in more detail later in section 5.9.

5.6 Optimisations

There are a few optimisations which may be included in the method of the previous section which reduce the size of the program returned quite considerably. We list them below:
First we should note that because of the fact that input nodes require zero to be placed in the coefficient $w_{(0,\delta)}$ and one to be placed in the coefficient $w_{(1,\delta)}$, where δ is the input node. Also because the degree starts at one or zero at the leaf nodes, gradually increasing as the SLP is ascended, there will be zero and one nodes to take into account.

- When considering plus nodes in the input program, an addition of zero is a null action and can be omitted.
- When considering times nodes there are two optimisations we can make:
  - a multiplication by zero will always be equal to zero, so these multiplications may simply be replaced by pointing to the zero element of the program.
  - a multiplication by one is a null action and can be omitted.
- Though if the above points are implemented the following optimisation will not make any difference to the length of the SLP, it would make a difference

to the time taken for its construction.
We record the degree of each node of the SLP as it is being constructed. This will give us an a-priori limit to the number of non-zero coefficients.

5.7 Division Removal (Introduction)

One of the main distinctions which may be made over the set of SLPs is whether they are *division free* or not. A division free SLP is an SLP which contains no division nodes.
When an SLP is evaluated, the division operation is the most expensive operation that must be performed, as it involves the calculation of an inverse, followed by a multiplication. We see that a division free SLP is preferable, so long as the length is not too much greater than the equivalent non-division free SLP .
The objects which we intend to represent by SLPs are polynomials that is they are objects which may be written in the following form:

$$p(x_1, \cdots, x_m) = \sum_{\{i_1, \cdots, i_m\}=0}^{n} c_i x_1^{i_1} \cdots, x_m^{i_m}$$

Because of this it is evident that it should not be necessary to include any division nodes in the representation. So with this fact in mind an obvious question is "how could division nodes occur in the first place". An answer is that there are certain algorithms which have steps translating to division nodes in the program, including our gcd algorithm. One example is the method used by Bareiss [1] to calculate the determinant of a matrix. We briefly describe the method in Appendix D. We have already seen an example of a non-division free SLP in section 1.3.

5.8 The Division Isolation Method

In the following method we combine any division operations and isolate them as a single division node at the end of the program. The method relies on the

following identities:

$$\frac{+}{-} \; : \; \frac{n_1}{d_1} \frac{+}{-} \frac{n_2}{d_2} \equiv \frac{n_1 d_2 \frac{+}{-} n_2 d_1}{d_1 d_2}$$

$$* \; : \; \frac{n_1}{d_1} * \frac{n_2}{d_2} \equiv \frac{n_1 n_2}{d_1 d_2}$$

$$/ \; : \; \frac{n_1}{d_1} / \frac{n_2}{d_2} \equiv \frac{n_1 d_2}{n_2 d_1}$$

This is very useful, though not essential, during division removal since we may use the following strategy. The first step is to use this method to isolate the divisions, then we use the method of section 5.9 to remove the final division node.

We develop the program which we wish to return, also we hold the numbers of lines which correspond to the numerator of a specific instruction in the input program (we denote these positions nPos and the corresponding instructions nProg) at the same time we hold the numbers of the lines which correspond to the denominator of specific instructions in the input program (we denote these positions dPos and the corresponding instructions dProg).
Given a sequence of operations $(+, -, *, /)$ to be performed on sparse form rational polynomials (quotients of sparse form polynomials, which are sums of power products), many traditional techniques for isolating the division operation would take gcds of the numerator and denominator at each stege, the gcds that were caslculated could then be divided out to keep the resulting quotient as small as possible. However in our case because our method is in effect a recording of the operations necessary to perform this gcd computation, this would be counterproductive as we would end up with a much larger straight line program.
The method that we use follows:

1. For each instruction, inst, do the following:
   (a) If inst is an input node or constant node then
      i. append inst to the return list
      ii. append the position of the instruction to numPos
      iii. append the position of 1 to denPos
   (b) If inst is an operation node, where $\circ_{inst}$ denotes the operation, and

$ptr_{(inst,1)}, ptr_{(inst,2)}$ denote the left and right pointers respectively; we have various possibilities

i. If $\circ_{inst}$ is $\stackrel{+}{-}$ then
append nodes corresponding to:

$$\text{nProg}.ptr_{(inst,1)} * \text{dProg}.ptr_{(inst,2)} \stackrel{+}{-} \text{nProg}.ptr_{(inst,1)} * \text{dProg}.ptr_{(inst,2)}$$

to the return program and append the position of the final instruction to nPos.
append the instruction corresponding to:

$$\text{dProg}.ptr_{(inst,1)} * \text{dProg}.ptr_{(inst,2)}$$

to the return program and append the position of this instruction to dPos.

ii. If $\circ_{inst}$ is $*$ then
append the instructions corresponding to

$$\text{nProg}.ptr_{(inst,1)} * \text{nProg}.ptr_{(inst,2)}$$

to the return program and append the position of the final instruction to nPos.
append the instructions corresponding to

$$\text{dProg}.ptr_{(inst,1)} * \text{dProg}.ptr_{(inst,2)}$$

to the return program and append the position of the final instruction to dPos.

iii. If $\circ_{inst}$ is / then
append the instructions corresponding to

$$\text{nProg}.ptr_{(inst,1)} * \text{dProg}.ptr_{(inst,2)}$$

to the return program and append the position of the final instruction to nPos.
append the instructions corresponding to

$$\text{dProg}.ptr_{(inst,1)} * \text{nProg}.ptr_{(inst,2)}$$

to the return program and append the position of the final instruction to dPos.

2. finally we append the instruction corresponding to

$$\text{last nProg / last Dprog}$$

to the program, this shall be the return node.

Though this method will always work, it may be usefull to check whether any division nodes exist in the first place, and if not just return the input program.

5.8.1 Optimisation

There are many cases when the denominator associated with an operation is one, this will be true for all of the operations which appear before the first division node as well as some that appear after it. This means that we may make optimisations by omitting any operations of the form $n*d$ where n is some node, and d is a node corresponding to one. In this case we append to the position list the position corresponding to n. If we have the case $d_1 * d_2$ where d_1 and d_2 are both one, then we append the position corresponding to one to the position list.

5.9 Newton approximation

In this section we describe the theory behind the algorithm given in subsection 5.5.1. We shall to a large extent be following Knuth [14]. Given that $f(x_1, \cdots, x_n)$ is a polynomial we may therefore express it without division. If however $f(x_1, \cdots, x_n)$ is presented to us as the quotient of two functions it would necessarily be an exact quotient. We may express it as $f(x_1, \cdots, x_n) = \frac{u(x_1, \cdots, x_n)}{v(x_1, \cdots, x_n)}$. We find that we may use *Newton approximation* to construct a polynomial univariate in one variable, whilst the coefficients are multivariate functions, possibly containing division, of all the other variables. Our approach will be to approximate $g(x_1, \cdots, x_n) = \frac{1}{v(x_1, \cdots, x_n)} \mod x_i^d$ where x_i is an arbitrary variable and d is the degree in that variable. Our initial approximation shall be to approximate $\frac{1}{v(x_1, \cdots, x_n)}$ by the coefficient of x_i^0, this will be the value arrived at by specialising the variable x_i to 0 that is $\frac{1}{v(x_1, \cdots, 0, \cdots, x_n)}$. Call this approximation α_0. Now we make a recursive definition for the sequence of Newton approximations. We shall show that to approximate $f(x_1, \cdots, x_n) \mod (x_i^d)$ it is only necessary to calculate $\lceil \log(d) + 1 \rceil$ members of this sequence. The definition of the sequence follows:

$$\begin{aligned}\alpha_0 &= g(x_1, \cdots, 0, \cdots, x_n) \\ \alpha_m &= 2\alpha_{m-1} - \alpha_{m-1}^2 v(x_1, \cdots, x_n)\end{aligned} \tag{5.19}$$

It is clear that $\alpha_0 = g(x_1, \cdots, x_n)(1 - x_i) \bmod x_i$. α_0 is the constant term in $g(x_1, \cdots, x_n)$ with respect to x_i. We should note however that so long as $n > 1$, that is the function $g(x_1, \cdots, x_n)$ has more than one variable, the program corresponding to the construction of α_0 will involve at least one division node.

Now assume that

$$\alpha_m = g(x_1, \cdots, x_n)(1 - x_i^k) \bmod x_i^k = g(x_1, \cdots, x_n) \bmod x_i^k$$

At the stage m, $k \geq 2^m$
From the definition of the sequence and the fact that $g(x_1, \cdots, x_n) = \frac{1}{v(x_1 \cdots x_n)}$ and since the x_i^k term is in fact x_i^k.

$$\begin{aligned}\alpha_{m+1} &= \left(\frac{2 - 2x_i^k}{v(x_1, \cdots, x_n)} - \frac{(1 - 2x_i^k + x_i^{2k}) v(x_1, \cdots, x_n)}{v^2(x_1, \cdots, x_n)} \right) \\ &= \frac{1 - x_i^{2k}}{v(x_1, \cdots, x_n)}\end{aligned}$$

Now $\alpha_{m+1} \bmod x_i^{2k} = \frac{1}{v(x_1, \cdots, x_n)} \bmod x_i^{2k} = g(x_1, \cdots, x_n) \bmod x_i^{2k}$

Now since we double the power of the modulus at every step, we can conclude that at the k^{th} step the polynomial represented by α_k is the approximation of $g(x_1, \cdots, x_n)$ accurate $\bmod(x_i^{2^k})$ in other words in order to be accurate $\bmod(x_i^{d+1})$ we need to perform the above iteration $\lceil \log_2(d) + 1 \rceil$ times. We then must only consider the first $d + 1$ coefficients. Using the above method, we may perform an interpolation of a polynomial containing division nodes. The Newton iteration technology has already been used in the iteration method of section 5.5.1 to deal with division nodes, we now discuss how we may use the technology to perform division removal from SLPs.

5.10 Division Removal Method

In this section we essentially describe an extension to the algorithm given in section 5.5.1. The algorithm will perform division removal with respect to every variable in the SLP. We note that Newton approximation as it stands will only perform division removal with respect to one variable. There may still be division nodes in the α_0 element of the sequence. To eliminate divisions entirely, construct a sequence of Newton approximations $\alpha_{(i)}^{(1)}, \cdots, \alpha_{(i)}^{(n)}$ s.t. $\alpha_l^{(j)} = \alpha_0^{(j+1)}$ and $\alpha_0^{(1)} = g(0, \cdots, 0) = \frac{1}{v(0, \cdots, 0)}$.
in the preceding $0 \leq i \leq \lceil \log_2(d_i) \rceil$ where d_i is the degree of x_i in $f(x_1, \cdots, x_n)$ and $0 \leq j \leq n$, also the terms $\alpha_l^{(j)}$ are the last element in the sequence $\alpha_i^{(j)}$.
We see that $g(0, \cdots, 0)$ is in the quotient ring of $\mathbf{R}$ where $\mathbf{R}$ is the base ring of $f(x_1, \cdots, x_n)$. If $\mathbf{R}$ is a field then this quotient ring is $\mathbf{R}$ itself, however if this is not so, then we must pay a price for obtaining a division free SLP, that is that the base ring of the division free SLP will be **Fraction(R)**.

1) Evaluate $v(0, \cdots, 0)$, this will be a value in $\mathbf{R}$.
Evaluate $\frac{1}{v(0, \cdots, 0)} \in$ **Fraction(R)**

2) Creation of $\frac{1}{v(0, \cdots, 0, x_n)} \bmod x_n^{d_n}$ where d_n is the degree of x_n in $f(x_1, \cdots, x_n)$.
Via. the Newton approximation sequence $\alpha^{(1)}$

- $\alpha_0^{(1)} = \frac{1}{v(0, \cdots, 0)} \bmod x_n$
- $\alpha_{m+1}^{(1)} = 2\alpha_m^{(1)} - (\alpha_m^{(1)})^2 v(0, \cdots, 0, x_n) \bmod x_n^{2^m}$
  $0 < m \leq \lceil \log_2(d_n) + 1 \rceil$

$\vdots$

i) Creation of $\frac{1}{v(0, \cdots, 0, x_{n-i+2}, \cdots, x_n)} \bmod \prod_{j=n-i+2}^{n} x_j^{d_j}$ where d_j is the degree of x_j in $f(x_1, \cdots, x_n)$. Via. the Newton approximation sequence $\alpha^{(i-1)}$.

- $\alpha_0^{(i-1)} = \frac{1}{v(0, \cdots, 0, x_{n-i+2}, \cdots, x_n)} \bmod x_{n-i+2} \prod_{j=n-i+3}^{n} x_j^{d_j}$
- $\alpha_{m+1}^{(i-1)} = 2\alpha_m^{(i-1)} - (\alpha_m^{(i-1)})^2 v(0, \cdots, 0, x_{n-i+2}, \cdots, x_n) \bmod x_{n-i+2}^{2^m}$
  $0 < m \leq \lceil \log_2(d_{n-i+2}) + 1 \rceil$

$\vdots$

n+1) Creation of $\frac{1}{v(x_1,\cdots,x_n)} \bmod \prod_{i=1}^{n} x_i^{d_i}$ where d_1 is the degree of x_1 in $f(x_1,\cdots,x_n)$. Via. the Newton approximation sequence $\alpha^{(n)}$

- $\alpha_0^{(n)} = \frac{1}{v(0,x_2,\cdots,x_n)} \bmod x_1 \prod_{i=2}^{n} x_i^{d_i}$
- $\alpha_{m+1}^{(n)} = 2\alpha_m^{(n)} - \left(\alpha_m^{(n)}\right)^2 v(x_1,\cdots,x_n) \bmod x_1^{2^m}$
  $0 < m \leq \lceil \log_2(d_1) + 1 \rceil$

The value returned in $\alpha_{\lceil \log_2(d_1)+1 \rceil}^{(n)}$ is $g(x_1,\cdots,x_n) \bmod \prod_{i=1}^{n} x_i^{d_i}$ we may multiply this by $u(x_1,\cdots,x_n)$ to obtain $f(x_1,\cdots,x_n) \bmod \prod_{i=1}^{n} x_i^{d_i}$ which will calculate the polynomial intended, involving no divisions.

A problem which may occur is, if $v(0,\cdots,0) = 0$ then its reciprocal does not exist. If this is the case then a failure will occur in step 1). There are two ways in which we could attack this problem. One approach would be to use a multivariate form of *l'Hôpital's* rule, one form of which states:

if $\lim_{x_1,\cdots,x_n \to 0} f(x_1,\cdots,x_n) = 0$ and $\lim_{x_1,\cdots,x_n \to 0} g(x_1,\cdots,x_n) = 0$
then $\lim_{x_1,\cdots,x_n \to 0} \frac{f(x_1,\cdots,x_n)}{g(x_1,\cdots,x_n)} = \lim_{x_1,\cdots,x_n \to 0} \frac{\nabla f(x_1,\cdots,x_n)}{\nabla g(x_1,\cdots,x_n)}$

$\nabla f(x_1,\cdots,x_n)$ is the gradient of f, that is the sum of the partial derivatives.
in terms of the SLP created to implement this alteration, we could use the method due to Kaltofen [4] to calculate the partial derivatives. The new SLP is length $O(nl)$ where l is the length of the denominator SLP, now it is required to evaluate the derivative SLPs at $(0,\cdots,0)$. We may again obtain a zero denominator necessitating a further l'Hôpital step, this may have the drawback that the resulting program would be very large. We therefore propose a further method which we have implemented that is the following.
We translate the polynomial by a vector $(\xi_1,\cdots,\xi_n)$, then apply the method of section 5.10, finally we must back translate by the same value. This is equivalent to performing the evaluation of $v(x_1,\cdots,x_n)$ at the point $(\xi_1,\cdots,\xi_n)$. We may guarantee that there exists a point at which $v(x_1,\cdots,x_n) \neq 0$ because $f(x_1,\cdots,x_n)$ is well defined, ie. the SLP is valid. It is desirable to minimise the number of translations necessary, as this will lead to a smaller SLP. We propose the following scheme for choosing points:

For variable x_i try values $(0, \cdots, 0, v, 0, \cdots, 0) \mid \begin{matrix} 1 \leq i \leq n, \\ 1 \leq v \leq d_i \end{matrix}$

where d_i is the degree of x_i in v. If every evaluation gives zero, which will not happen very often as it implies that f is identically zero along every axis, we instead try translating every variable by a random quantity. It would be possible to perform a combinatorial search for an evaluation point which gave a non-zero v, and so perform the minimum number of translations, however it was thought that the size of the saving would not justify the complexity of code required to perform the point traversal. The increment to the size of the SLP is $O(1)$ in the case where we translate one variable, it is $O(n)$ if we translate every variable. We note that translation is a cheap operation when using SLP representation, as we must only add a few nodes to the base of the DAG.

5.11 Complexity Analysis

We shall denote the proportion of plus or minus nodes (times nodes, quotient nodes, input nodes and constant nodes) in the program by $D_{\pm}$ (D_* , $D_/, D_I, D_C$ respectively). $D_{\pm}$ denotes the proportion of either plus or minus nodes.
We first analyse the complexity of the division isolation method of section 5.8. We shall consider the contributions made by each type of node to the size complexity of the programs returned by this method.

- Constant Nodes and Input Nodes:
  For every node of these types a single node must be added to the returned program.
- Plus or minus nodes:
  For every node of these types three times nodes and one plus or minus node must be added to the return program.
- Times Nodes or quotient nodes:
  For every node of these types two multiplication nodes must be added to the return program.

The overall size of the new program will be $(D_I + D_C + 4D_{\pm} + 2D_* + 2D_/)l$ that is the asymptotic complexity is linear in l.

We now analyse the complexity of the interpolation method of section 5.5. We denote the degree of the polynomial in the interpolation variable by d. We shall consider the contributions made by each type of node to the size complexity of the programs returned by this interpolation method.

- Plus or minus nodes:
  For one node we must construct a new node for each of the $d+1$ coefficients, there are $D_{\pm}l$ nodes, where l is the program length. So the contribution to the new program has length $O(D_{\pm}ld)$.

- Times nodes:
  For one node we must initially construct $\frac{d^2}{2}$ times nodes, and then construct $\sum_{i=0}^{d} \log_2(i)$ plus nodes. So the contribution to the new program has length $O(D_*l(\frac{d^2}{2} + \sum_{i=0}^{d} \log_2(i))) = O(D_*ld^2)$.

- Quotient nodes:
  For a quotient node we use the method in section 5.9. The first step is to create a program representing the specialisation of x_1 to 0. This program will have length $O(l)$. The following steps require the creation of program representing the polynomials α_i $0 < i \leq \lceil \log_2(d) + 1 \rceil$ in equations 5.19. There are clearly $\lceil \log_2(d) + 1 \rceil$ of these steps, each of which requires:

  - multiplying each coefficient by two, that is d operations.
  - raising the previous element in the sequence to the power two, that is $\frac{d^2}{2} + \sum_{i=0}^{d} \log_2(i)$ operations.
  - multiplying the previous *square* by the denominator, again $\frac{d^2}{2} + \sum_{i=0}^{d} \log_2(i)$ operations.
  - an addition, that is d operations.

  In total that is $2d + 2\frac{d^2}{2} + \sum_{i=0}^{d} \log_2(i) = O(d^2)$ operations per stage of the sequence.
  We may conclude that the contribution to the new program will have length $D_/l\lceil \log_2(d) + 1 \rceil(2d + 2\frac{d^2}{2} + \sum_{i=0}^{d} \log_2(i)) = O(D_/l\lceil \log_2(d) + 1 \rceil d^2)$

We could use an alternative technique in which we isolate the quotient node initially, then we apply the Newton iteration to that node, there will now only be one division node, making a contribution of $O(l\lceil\log_2(d)+1\rceil d^2)$ operations, however the rest of the program will have been increased in length, by a factor of up to four, where the number of times nodes has been increased by up to three and the number of plus or minus nodes stays the same.

The overall size of the new program will be $O(D_+ld)+D_*ld^2+D_/l\lceil\log_2(d)+1\rceil d^2)$ or using the alternative technique $O(D_+ld)+(3D_+ +2D_*)ld^2+l\lceil\log_2(d)+1\rceil d^2) = O(l\lceil\log_2(d)+1\rceil d^2)$.

We now analyse the complexity of the division removal method of section 5.10. We require the partial evaluations $f(0,\cdots,0),\ \cdots,f(x_1,\cdots,x_n)$. A naive implementation will require a program with $O(ln)$ nodes. We have designed a method which constructs a program which represents equivalent objects, in $2l = O(l)$ nodes. The rest of the program will have length $\sum_{i=1}^{n} 4\lceil\log_2(d_i)+1\rceil$.
The asymptotic complexity is therefore $O(l+n\log_2(d_m))$ where d_m is the maximum of the degrees.

5.12 Conclusion

In this chapter we have presented various interpolation techniques, together with related ideas. These are important steps in our gcd algorithm, and necessary for implementing efficient algorithms using SLP representation.

Chapter 6

Gcd strategies

We shall consider two Gcd strategies which we have implemented. The first technique we consider commences with an interpolation method which does not require any special elements to be in the base field (for example n'th roots of unity). The interpolations considered are the Lagrangian interpolation described in section 5.2 and the direct interpolation technique we describe in section 5.5. These allow us to use our domain StraightLineValue. The second strategy commences with the FFT Interpolation of the SLP. This interpolation introduces modular values within the program. This forces us then to use our second implementation, StraightLineValueMod, which considers the constant nodes to be in a modular field, having profound implications on the subsequent algorithm.

6.1 Pseudo-Remainder

A major part of the following gcd algorithms, is the *euclidean loop.* During every iteration the size of the program will be increased by some amount, the reason for this is because a *pseudo remainder* must be encoded in the program, for every iteration. We describe this pseudo-remainder algorithm. The pseudo-remainder is similar to the *remainder* under normal *division.* The pseudo-remainder is the remainder after *pseudo-division* of two polynomials, pseudo-division is like division except that no divisions over the base ring are required, this is desirable for SLPs as it infers that no division nodes must be added. We describe the

algorithm for finding the pseudo-remainder of two polynomials u and v, with degrees d_u and d_v respectively.

6.1.1 The Pseudo-Remainder Algorithm

u and v must be represented in the mixed representation of section 5.2.1. We shall denote the coefficients of u with respect to $x_0, \cdots, x_{d_u}$ as $u_0, \cdots, u_{d_u}$ respectively and the coefficients of v with respect to the coefficents of $x_0, \cdots, x_{d_v}$ as $v_{(0,0)}, \cdots, v_{(0,d_v)}$ respectively. The algorithm follows:

1. calculate the number of iterations required, that is $k = d_u - d_v$
   If $d_u > d_v$ return v.
2. for j in 0 to k, repeat the following steps:
   (a) add times nodes to the program which correspond to the operations, $t_{(1,i)} = v_{j,i} * u_{d_u}$ for i in 0 to $(d_v - k - 1)$.
   This corresponds to multiplying v_j (that is the polynomial which has $v_{(j,i)}$ as the coefficient of x_i) by the leading coefficient of u.
   (b) add times nodes to the program corresponding to the operations, $t_{(2,i)} = u_i * v_{(j,d_v-k)}$ for i in 0 to $(d_u - 1)$.
   This corresponds to multiplying u by the leading coefficient of v_j.
   (c) add subtraction nodes to the program corresponding to the operations, $v_{(j+1,i+k-j)} = t_{(1,i)} - t_{(2,i+k-j)}$ for i in 0 to $d_u - 1$.
   This corresponds to subtraction of step 2b from step 2a.
3. The pseudo-remainder is then returned as the coefficients $v_0, \cdots, v_{d_u}$.

This algorithm results in a sequence of multiplications and subtractions to be added to the SLP.

Example

Consider the calculation of the pseudo-remainder of $6x + z + 3$ after division by $2x + 1$. We shall denote $2x + 1$ by p, and $6x + z + 3$ by q.

In traditional sparse form notation the algorithm performs the following steps:

$2x+1\overline{)6x+\ z+3}$

step 1	$12x+2z+6$	multiplication by the leading coefficient of p
step 2	$12x+6$	multiplication by the leading coefficient of q
step 3	———————	subtraction step

$\cdot \quad 2z+6-6=2z$

We consider the SLP which denotes p and q in the mixed representation of section 5.2.1:

line 1 : Constant node : 1 $>_{p_0}$
line 2 : Constant node : 2 $>_{p_1}$
line 3 : Constant node : 3
line 4 : Constant node : 6 $>_{q_1}$
line 5 : Input node : z
line 6 : Operation node : (plus line 3, line 5) $>_{q_0}$

We must now add nodes to record the calculation of the pseudo-remainder procedure,

step 1 : multiplication by the leading coefficient of p:

line 7 : Operation node : (times line 6, line 2)

step 2 : multiplication by the leading coefficient of q:

line 8 : Operation node : (times line 1, line 4)

Note that we do not need to calculate the product of the leading coefficients of p and q, because we are garuanteed that they will cancel in the following subtraction step.

step 3 : subtraction of the results from step 2 from the results of step 1:

line 9 : Operation node : (minus line 7, line 8)

6.2 Basic Gcd Strategy

1. Bound calculation

   (a) get a bound for the maximum quantity encountered during any evaluation which will occur during the calculation.
   First calculate a coefficient bound, using the technique presented in, section 4.8, denote this bound c_b.
   Now we calculate a bound to any evaluations of these objects.
   This will be given by the formula:

$$b := \left(\frac{x_{max}^{d+1} - 1}{x_{max} - 1}\right)^n c_b$$

   where:

   i. x_{max} is the maximum evaluation point encountered.

   ii. d is the degree in the leading variable.

   iii. n is the number of variables.

   (b) calculate a set of primes $\{p_1, \cdots, p_m\}$ such that $\prod_{i=1}^{n} p_i > 2b$

2. Removal of content: in the following $\rho \in \{1, 2\}$

   (a) get the translation vector, this is constructed from random values, $(b_2, \cdots, b_n)$ taken from $\mathbf{Z}^{(n-1)}$.

   (b) perform the translation $\tilde{f}_\rho(x_1, y_2, \cdots, y_n) = f(x_1, y_2 + b_2 x_1, \cdots, y_n + b_n x_1)$. By making the substitutions:

$$x_2 \leftarrow y_2 + b_2 x_1 \quad , \quad \cdots \quad , \quad x_n \leftarrow y_n + b_n x_1$$

   (c) get the coefficients of $\tilde{f}_\rho$ denoted by $\tilde{f}_{\rho,1}, \cdots, \tilde{f}_{\rho,i}$ using an interpolation which works over Z, e.g. Lagrange interpolation or direct interpolation as detailed in 5.5.

   (d) find the leading coefficient of $\tilde{f}_\rho$ in x_1,
   if the leading coefficients of both $\tilde{f}_1$ and $\tilde{f}_2$ are non-constant then the content may be as well, so we must try a different translation, repeat from step 2a.

N.B. we note that with high probability this implies that we only have to do the constant test once.

3. euclidean loop

   (a) $r :=$ pseudo-remainder$(\tilde{f}_1, \tilde{f}_2)$
   (the pseudo-remainder is performed using the method of section 6.1).

   (b) $\tilde{f}_2 := \tilde{f}_1; \tilde{f}_1 := r$
   if r is not 0 (that is degree of r $\neq -1$) goto step 3a

   (c) r is now the gcd $\in \mathbf{Z}(y_2 \cdots, y_n)[x_1]$

6.3 Modular Strategy

We must calculate a bound for the coefficients of the gcd at this stage, because the translation constants which we calculate are required to be in a modular field where the modulus is greater than twice this bound, to allow for negative numbers.

1. Bound calculation

   (a) get a bound for the coefficients of the gcd, this may be calculated using the formula:

$$b := 2^{\min\{td := tdeg\ f_\rho\}} \min\{\sqrt{nf_{max}}\}$$

   where f_{max} is an upper bound for the coefficients of $f_{\rho,i}$, tdeg is the *total degree*(where total degree is $\sum_{i=1}^{n} d_i$ and d_i are the degrees with respect to each variable in $f_{\rho,i}$) of the polynomial, we use a bound for this, viz. the Total SLP-degree described in section 3.13, this may be easily calculated for an SLP.
   The *tdeg* of a polynomial will also be a bound for the degree of the translated polynomial in the leading variable.

   (b) calculate a set of primes $\{p_1, \cdots, p_m\}$ such that $\prod_{i=1}^{n} p_i (=: P) > b$
   these primes must also satisy the following condition:
   They must be Fourier primes, these are primes, p_f s.t. the modular fields $\mathbf{Z}_{p_f}$ contain n, n^{th} roots of unity (see section 5.3.1) for some n.

2. Removal of content: in the following $\rho \in \{1, 2\}$

   (a) get the translation vector, this is constructed from random values, $(b_2, \cdots, b_n)$ taken from $\mathbf{Z}_{p_f}^{(n-1)}$.

   (b) perform translation $\tilde{f}_\rho(x_1, y_2, \cdots, y_n) = f(x_1, y_2 + b_2 x_1, \cdots, y_n + b_n x_1)$. By making the substitutions:

$$x_2 \leftarrow y_2 + b_2 x_1, \quad \cdots, \quad x_n \leftarrow y_n + b_n x_1$$

   (c) get the coefficients of $\tilde{f}_\rho$ denoted by $\tilde{f}_{\rho,1}, \cdots, \tilde{f}_{\rho,i}$ using asymptotically fast interpolation (see section 5.3) performed in the field $\mathbf{Z}_{p_f}$.
   We note that the coefficients will be *modular* SLPs, due to the partial evaluations performed at the modular roots of unity.

   (d) find the leading coefficient of $\tilde{f}_\rho$ in x_1,
   check whether the leading coefficient is in $\mathbf{Z}_{p_f}$, or in some polynomial extension
   if the leading coefficients of both $\tilde{f}_1$ and $\tilde{f}_2$ are non-constant then the content may be as well, so we must try a different translation, repeat from step 2a.
   N.B. we note that with high probability this implies that we only have to do the constant test once.

3. Euclidean Loop

   (a) Set $gcd\_so\_far := 1$

   for each p_i where $(1 \leq i \leq m)$ do:

   Working in $\mathbf{Z}_{p_i}$

   (b) $r :=$ pseudo-remainder$(\tilde{f}_1, \tilde{f}_2)$
   (the pseudo-remainder is performed using the method of section 6.1).

   (c) $\tilde{f}_2 := \tilde{f}_1; \tilde{f}_1 := r$
   if r is not 0 (that is degree of r $\neq -1$) goto step 3b

   (d) r is now the gcd $\in \mathbf{Z}_{p_i}(y_2 \cdots, y_n)[x_1]$

   (e) check for unlucky prime moduli by checking that:

$$\deg\ r \leq \deg\ gcd\_so\_far$$

if the test shows an unlucky prime goto step 1b, however making sure that the primes selected are new (Notice that this is unlikely to happen as unlucky primes have a sparse occurence).
Otherwise update gcd_so_far by performing the chinese remainder algorithm using the value r with the modulus ρ_i and perform the next euclidean loop.

The final step is common to both strategies, though of course the objects will be in the relevant domain.

6.4 Calculate Gcd

It is necessary to maintain the monotonicity of the result, so we must now divide every coefficient by the leading coefficient. In the following d_x is the degree of the gcd in the leading variable.

1. divide every coefficient by the leading coefficient, to obtain a new ordered set of coefficients: $\{r_1 := 1, r_2, \cdots, r_{d_x}\}$.
2. Reconstruct Polynomial
$$\tilde{gcd} := \sum_{i=0}^{d_x} r_i x^i$$
3. Perform the back translation:
$$gcd(x_1, \cdots, x_n) := \tilde{gcd}(x_1, x_2 - b_2 x_1, \cdots, x_n - b_n x_1)$$
By making the substitutions:
$$y_2 \leftarrow x_2 - b_2 x_1 \quad , \quad \cdots \quad , \quad y_n \leftarrow x_n - b_n x_1$$

We thus obtain the gcd of f_1 and f_2.

6.5 A Monte Carlo Gcd Algorithm

We have also implemented a Monte Carlo version of the algorithm of section 6.2. The inherent probability of correctness allowed must be sent as a parameter to the domain. Also the preemptive probability of incorrectness for each object is stored as part of the representation for that object.
Every time a gcd is performed on two objects we must calculate the absolute probability of incorrectness of the answer. This will consist of two parts:

- The new probability of incorrectness due to Monte Carlo equality tests, there are two of these which count. The constant tests to ensure that the content has been removed and the equality test for exiting the Euclidean Loop. We should note that the probability of incorrectness for an equality test which returns false is zero.
  In the following p_i is the implicit probability of incorrectness allowed. We shall calculate the probability of incorrectness with which each equality must be performed. We may assume that both the tests are independant, as this will allow us to calculate an upper bound on the probablity of incorrectness, we shall denote this p_{eq}, we may use the following formula:

$$p_i = 2p_{eq} - p_{eq}^2 \tag{6.1}$$
$$0 = -p_{eq}^2 + 2p_{eq} - p_i \tag{6.2}$$

  We may solve this for p_{eq}:

$$p_{eq} = \frac{-2 \overset{+}{-} \sqrt{4 - 4p_i}}{-2} \tag{6.3}$$
$$= 1 \overset{-}{+} \sqrt{1 - p_i} \tag{6.4}$$

  now p_{eq} must be in the interval $[0, 1]$, therefore

$$p_{eq} = 1 - \sqrt{1 - p_i} \tag{6.5}$$

- The preemptive probability of incorrectness (p_m) for both arguments (p_1, p_2). This is calculated using the following formula:

$$p_m = p_1 + p_2 - p_1 p_2 \tag{6.6}$$

The probabilities p_i and p_m must be combined to give the final absolute probability (p_f), which is returned with the gcd SLP as part of the return value. We use the following formula:

$$p_f = p_m + p_i - p_m p_i \tag{6.7}$$

We use the AXIOM domain Float to represent the probabilities. However we have a problem when evaluating the expression given in 6.5 because the precision may not be accurate enough to show any significant figures to this evaluation. Our solution assumes that we have some significant figures in p_i. If p_i is very small the calculation of p_e may result in 0.0, this causes infinite loops in certain Monte Carlo algorithms, including the equality test we perform, see section 4.14. We may avoid this problem by doubling the precision before performing the calculation, this will guarantee that p_e has some significant figures. In order that we do not constantly increase the precision (the precision would become exponentially accurate in the number of applications of the gcd operation), we must reset the precision to its initial value at the end of the gcd algorithm.

6.6 Algorithm Correctness

6.6.1 Bound calculation

We calculate a bound for the evaluation of a polynomial $f(x_1, \cdots, x_n)$ at a point $(\xi_1, \cdots, \xi_n)$
let $\xi_{max} = \max\{\xi_1, \cdots, \xi_n\}$
let $d_1, \cdots, d_n$ be the degrees in $x_1, \cdots, x_n$ respectively and $d_{max} = \max\{d_1, \cdots, d_n\}$.
Then, we may wright $f(\xi_1, \cdots, \xi_n)$ as the sum of monomials:

$$f(\xi_1, \cdots, \xi_n) = c_p \xi_1^{d_1} \cdots \xi_n^{d_n} + \cdots + c_0$$

where $p = \prod_{i=1}^{n}(d_i + 1)$. Let $c_{max} = max\{c_0, \cdots, c_p\}$ then:

$$\begin{aligned} f(\xi_1, \cdots, \xi_n) &< c_{max}(\xi_1^{d_1} \cdots \xi_n^{d_n} + \cdots + 1) && (6.8) \\ &= c_{max} \prod_{i=1}^{n} (\xi_i^{d_i} + \cdots + \xi_i + 1) && (6.9) \end{aligned}$$

$$< c_{max}'(\xi_{max}^{d_{max}} + \cdots + \xi_{max} + 1)^n \tag{6.10}$$

$$= c_{max}\left(\frac{\xi_{max}^{d_{max}+1} - 1}{\xi_{max} - 1}\right)^n \tag{6.11}$$

6.6.2 Removal of content

We notice that the gcd of two multivariate polynomials p, q is given by the following formula:

$$\gcd(p, q) = \gcd(\text{cont}(p), \text{cont}(q))\gcd(\text{pp}(p), \text{pp}(q))$$

The Euclidean algorithm will give a constant multiple of the gcd(pp(p)). It is therefore necessary to remove the content from f_1 and f_2 in order to calculate the gcd rather than a common divisor of lesser degree. As we are using SLP representation we notice that a good method is to translate by a random amount which will have a small effect on the size of the SLP; represented in sparse form, a polynomial will become very dense on translation. Our method is then to check whether the leading coefficient is constant, if so we know that the content must be constant and as such may be ignored (we are calculating gcds up to constant multiples). It is only with small probability that this condition will not be satisfied.

6.6.3 Calculation of Gcd

The result returned from the euclidean loop (steps 3c, 3e) is the gcd of $\tilde{f_1}$ and $\tilde{f_2}$ in $\mathbf{Z}(y_2, \cdots, y_n)[x_1]$. We must now divide by the leading coefficient of this gcd, this makes the polynomial monic with respect to x_1. Now an argument based on Gauss's lemma (which states that products of primitive polynomials are primitive) implies that this gcd is the gcd of $\tilde{f_1}$ and $\tilde{f_2}$ in $\mathbf{Z}[y_2, \cdots, y_n, x_1]$. Now since the translation mapping is a ring isomorphism, the back translation will give us the gcd of f_1 and f_2 over $\mathbf{Z}[x_1, \cdots, x_n]$.

6.7 Other Gcd techniques

6.7.1 Resultant based technique

An alternative technique, expounded by Loos [16] for calculating the gcd of polynomials is based on resultants and sub-resultant theory, this also suffers from the need to calculate the content of the polynomials. It would of course be possible for us to remove the content by translation as above.

6.7.2 Divide and conquer technique

A method originally due to Shönhage, then extended to hold in all euclidean domains by Moenk [20], also cited by Czapor [28] has been preposed. The method is a divide and conquer technique based on the extended euclidean algorithm, it thus has assymptotic advantages over the basic euclidean algorithm.

Chapter 7

Results

7.1 Performance Metrics for SLPs

Three important measurements concerned with the performance of an SLP algorithm are:

- The time taken to create the SLP
- The *depth* of the SLP created. Depth is defined in section 1.4.
- The *length* of the SLP created. Length is defined in section 1.4.

7.2 Interpolation Results

The interpolation process is a very important step in the gcd algorithms described in the previous chapter as it is necessary for converting from a purely SLP representation to the mixed representation described in section 5.2.1. We shall compare and contrast the interpolation algorithms described in chapter 5. The following graph shows the time taken to perform these algorithms on SLPs which represent polynomials of a range of degrees:

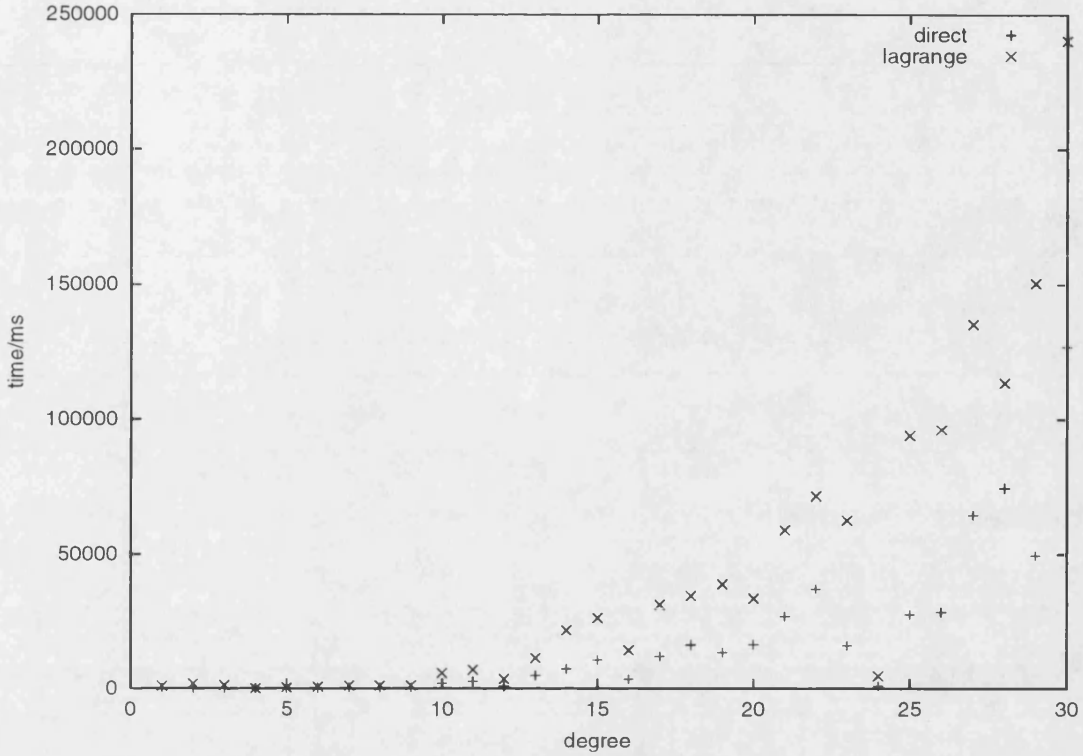


Figure 7.1: Graph demonstrating the superiority of the direct interpolation over the Lagrange interpolation

We conclude from this that the direct interpolation algorithm of section 5.5 is fastest in most situations, especially as the degree becomes large.

7.3 Gcd Results

We consider the three measures enumerated in 7.1 above. The values of these measures will depend on various attributes of the problems being considered, the most important being:

1 The SLP-degrees (see section 3.13) of the argument slps in the different variables and the number of variables

2 The total SLP-degrees (see section 3.13) of the argument slps.

3 The degree of the gcd of the argument polynomials.

4 the length of the input slps.

We shall now describe how and why these specific attributes will effect the algorithm being performed.

1 It will be necessary to perform equality tests at various points in the algorithm most notably in the degree checking at the start of every euclidean loop. This check will have a size $O(\prod_{x_i \in \mathbf{x}} \deg_x)$ where $\mathbf{x}$ is the set of variables in the SLP, in the deterministic case, though the check will only approach this value in the affirmitive case.

2 The degree of the translated polynomials in the major variable is determined by the total degree of the argument polynomials, which is bounded by the SLP-total degree of the argument SLPs. This will determine a bound on the length of the euclidean loop, as this degree gives a bound to the number of repetitions necessary.

3 The degree of the gcd will effect when the euclidean loop will terminate. If the degree of the gcd is small relative to the argument polynomials then the euclidean loop will be executed more times. However the final equality which will be positive (i.e. will return true) will not be very expensive. However if the degree of the gcd is similar to the degree of the argument polynomials, then even though the euclidean loop is only executed a few times, the final equality will have large degrees in it and therefore will be expensive.

4 There are various ways in which the length of the argument SLPs will effect the complexity of the gcd algorithm.

- **a** The *base* length of the gcd SLP will depend on the lengths of the argument SLPs. That is the length of the program which represents the coefficients of the polynomials which are input to the euclidean loop.
- **b** The time to perform the translations and the interpolation will depend on the length of the arguments.
- **c** The time to perform each evaluation is effected by the number of operations which must be evaluated. Also the size of the dummy lisp program which must be constructed and the time taken to perform its evaluation will be effected by the size of the program.

The following graph displays the correspondance between time against the product of the maximum of the degrees and the difference between the degree of the gcd and the degrees of the input polynomial. Each band corresponds to a different number of iterations of the euclidean loop (step 2, of the gcd algorithm described in section 6.2 that we use):

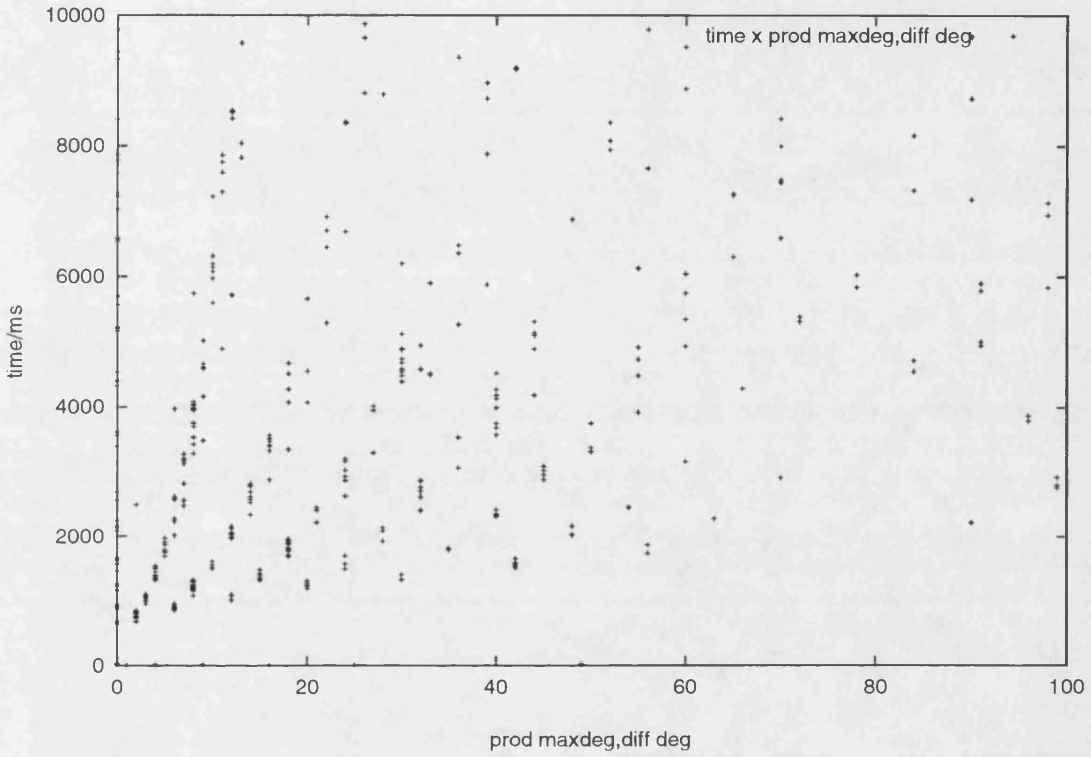


Figure 7.2: Graph displaying a banding effect due to varying numbers of iterations of the euclidean loop

In the following graph we show the superiority of the SLP representation as the degree of the gcd becomes greater. In this graph we are calculating the gcds of polynomials with three variables represented by SLPs. These SLPs are constructed to have a varying number of factors. We note that the size of the SLP has a profound influence over the time taken by the algorithm. The ratio measured on the y-axis indicates the quotient of the time taken by our Monte Carlo technique with respect to the deterministic technique which uses the sparse form used by AXIOM currently.

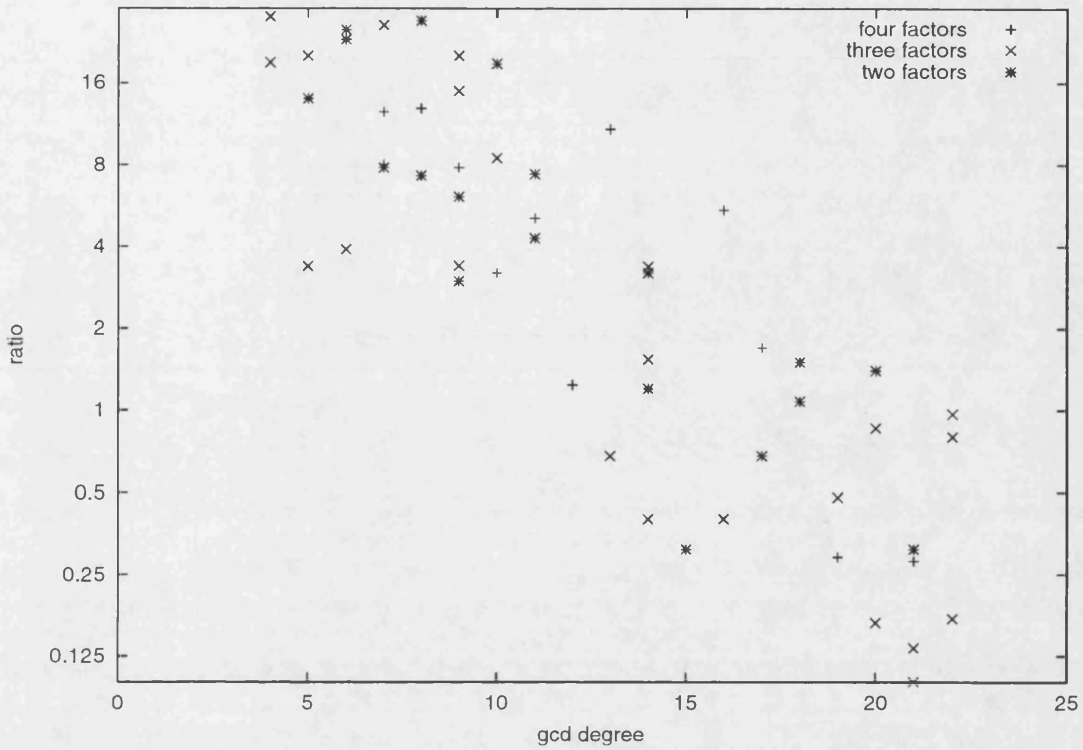


Figure 7.3: Graph to display the superiority to the SLP representation as the degree of the gcd becomes greater

7.4 Gröbner Basis Results

Gröbner basis were first introduced by B. Buchberger, under the direction of W. Gröbner. They comprise a generating set for a *polynomial ideal*, with some extra properties. Gröbner basis provide a good general technique for solving systems of multivariate non-linear equations (see Kaltofen [12]). We may use an algorithm due to Buchberger to calculate Gröbner basis, known as *Buchbergers Algorithm*. This involves polynomials known as *S-polynomials*. For two polynomials f_1, f_2, their S-polynomial is defined as

$$S(f_1, f_2) = h_1 f_1 - h_2 f_2$$

where

$$h_1 = \frac{\text{lcm}(\text{init } f_1, \text{init } f_2)}{\text{init } f_1}, \quad h_2 = \frac{\text{lcm}(\text{init } f_1, \text{init } f_2)}{\text{init } f_2}$$

In the above:

- init denotes the initial term of the polynomial (its argument) with respect to some ordering on terms, e.g. the *deglex* ordering, this is an ordering which depends first on the variables and then the degrees of a term.
- lcm denotes the Least Common Multiple of two polynomials, this satisfies the following relation:

$$\text{lcm}(f_1, f_2) = \frac{f_1 f_2}{\text{gcd}(f_1, f_2)}$$

As detailed in 1.6 we may use the SLP gcd algorithms in order to calculate a Gröbner basis in AXIOM.

The problem with using SLPs for calculating Gröbner basis using Buchbergers Algorithm is that Buchbergers Algorithm involves many gcd calculations of *small* poloynomials, i.e. polynomials with small degrees and containing a small number of variables. These calculations are not very efficient using our methods.

Chapter 8

Conclusion

8.1 Equality checking

Because of the non canonicality of the SLP representation, a deterministic equality check for these objects necessarily relies on the mathematical properties of the objects being represented rather than some syntactic check. It is claimed in [29] that for polynomials this problem is exponential in the number of variables in the polynomial. Indeed for polynomials represented in sparse form the number of terms increases exponentially with the number of variables. If we intend to produce algorithms which are sub-exponential in complexity, using current means we are forced to use certain non-deterministic equality checking methods, for example see section 4.14.

8.2 Modular Viewpoint

We discuss the problems that occur when we consider the modular viewpoint in a categorical setting. Consider the Gcd Domain described in section 1.6. Here we have a problem, as when the gcd function is applied to different SLPs, the bounds calculated and hence the moduli of the resulting SLPs may be different. It will not be valid to then do operations between these SLPs, as they have different moduli. One may argue that if an SLP had a modulus which was lacking some

factors it would be possible to retrospectively calculate the constants with respect to the necessary moduli i.e. the lacking factors, however it is unlikely that this would be trivial to implement in a generic setting.

8.2.1 A possible solution

We can resolve the problem of the moduli of programs p_1 and p_2 being different. Let m_1, m_2 denote the moduli of p_1 and p_2 respectively, we may factor m_1 and m_2 to get factors $fm_{1,1}, \cdots, fm_1, n$ and $fm_{2,1}, \cdots, fm_2, m$. Some of these factors may be equal, assume WLOG that these are the first l factors, so:

$$fm_{1,1} = fm_{2,1}, \cdots, fm_{1,l} = fm_{2,l}$$

If we can construct slps representing the same object as p_1 with moduli $fm_{2,l+1}, \cdots, fm_2, m$ and as p_2 with moduli $fm_{1,l+1}, \cdots, fm_1, n$.Then we may produce programs $\tilde{p_1}$ and $\tilde{p_2}$ with modulus

$$\prod_{i=1}^{l}(fm_1, i) \prod_{i=l+1}^{n}(fm_1, i) \prod_{i=l+1}^{m}(fm_2, i) = lcm(m_1, m_2)$$

$\tilde{p_1}$ and $\tilde{p_2}$ have the same modulus, and it is therefore valid to perform operations between them.
In order to produce the new slps it is necessary that we have some means for calculating the modular constants which appear in the program. We propose that every constant should have a function associated with it which implements its construction.

8.3 Object Oriented Viewpoint

AXIOMs categorical structure is in effect an *object oriented* structure, with Categories as objects on one level and Domains as objects on another. This structure allows us to take 'short cuts' when coding an algorithm. However it does mean that we cannot make use of any prior knowledge that we have of some feature of the algorithm. This may lead to nonoptimal code. If we wish to make use of these features to perform some optimisations, the resulting code is often badly

structured and verbose.

Example:

We shall look at the example of finding the *pseudo-remainder* of p_1 with respect to p_2. Where p_1 and p_2 are polynomials each represented by an array of SLPs, with each SLP representing a coefficient with respect to one distinguished variable. We first look at a naïve way of using the built in AXIOM function pseudoRemainder($\cdot$, $\cdot$) to perform the calculation.

1. form a UnivariatePolynomial(x, SLP(R)) for each polynomial, denote these as u_1, u_2 respectively, in this x is the distinguished variable and R is the base ring for p_1, p_2.
2. make the function call pseudoRemainder(u_1,u_2)
3. reconstruct a polynomial from these coefficients, or alternatively return an array of programs, where each program is a coefficient.

The function pseudoRemainder($\cdot$, $\cdot$) necessarily uses the arithmetic operations defined in SLP(R), this will involve merging programs together at each application. It is far more efficient to construct a list of operations which code the pseudo-remainder algorithm and append this to the end of the program which represents the argument polynomials. Another advantage to this is that we could share the same program between all the coefficients, each one being represented by an SLP which has a different return instruction from the same program.

8.4 Monte Carlo Viewpoint

The algorithm of section 6.5 is a Monte Carlo algorithm. The problem of embedding this sort of algorithm in AXIOM has been elucidated in section 3.17. We would like to produce a system with the same categorical structure as AXIOM which allows Monte Carlo algorithms to be used. In such a system it would be necessary to guarantee that results were correct to a specific probability, i.e. we would require that the inherent probability was zero. We give an example to illustrate the sort of algorithms we might use:

Example:

We shall consider the case where we have a Monte Carlo Matrix domain, built over a Monte Carlo SLP domain. We shall look at the case of a rank function, that is a function to calculate the rank of a matrix (the number of linearly independant rows or columns). This may be achieved by reducing the matrix to *Row Echelon Form* and then counting the number of non-zero rows. A maximum of n^2 equality tests must be performed, where n is the size of the matrix. So we see that if we can find the solution to the equation defined by the recursive relation:

$$\begin{aligned} f_1 &= p \\ f_m &= f_{m-1} + p - f_{m-1}p \mid m > 1 \end{aligned}$$

(or a lower bound to the solution)
where f_n is the probability with which we require to limit the probability of an incorrect answer to the rank problem. The Monte Carlo operation we are concerned about in the SLP domain is the equality check, we denote this by eq_{SLP}. The solution to the equation defined above (p) is the limit to the probability of incorrectness that we send to eq_{SLP}. So the *rank* command in the Monte Carlo system would need to calculate p. Now on each application of eq_{SLP}, p is an upper bound to the probability of an incorrect answer.

8.5 Expression Swell

Expression swell is a problem which may occur in mathematics. When dealing with sparse form polynomials, expression swell manifests itself in that intermediate expressions, (expressions which it is necessary to calculate in order to determine the answer, but which do not themselves make up part of the answer) may become very large even though the answer is small.
Example:
A classic example used by many authors ([9],[14],[30]) to show the problems of expression swell is the following,
find the gcd of the following polynomials:

$$\begin{aligned} u(x) &= x^8 + x^6 - 3x^4 - 3x^3 + 8x^2 + 2x - 5 \\ v(x) &= 3x^6 + 5x^4 - 4x^2 - 9x + 21 \end{aligned}$$

If we solve this problem using the naïve euclidean algorithm with pseudo-division we must calculate a polynomial remainder sequence, the last element of which is a constant with 35 decimal digits. The information this imparts is that the two polynomials are relatively prime. This information may be stored in one bit.

Even though in some cases SLPs may reduce expression swell by reducing the sizes of polynomial objects, they do not remove it and indeed some objects which in sparse form may be very simple may be represented by SLPs which are relatively large due to cancellations which may occur.
Example:
If two polynomials are relatively prime their gcd will be identically unit. However the SLP formed to represent this gcd could be very large.

8.6 Black Box Representation

We look at a further representation which may be used for representing polynomials called *Black Box representation*, see [3, 13]. In this representation polynomials are represented as *Black Boxes*: a Black Box is an object which takes as input a value for each variable, and then produces the value of the polynomial at the specified point. We note that Black Boxes may in fact be modeled by SLPs, together with a complete evaluation function. However the Black Box model is more general, in that the implementation 'inside' the Black Box may include loops and conditional tests.
Example:
If an SLP is formed to represent the gcd of two polynomials using the algorithm of section 6.2, then a section is added to the program for every time that the euclidean loop is executed. However using Black Box representation the loop would be represented explicitly. In effect we would have a *gcd Black Box* which would make *oracle calls* to Black Boxes which represent the polynomials. this Black Box would be a representation for the gcd of the polynomials.

We should note that the interpolation algorithm of section 5.5 could not be used on Black Box objects, as it relies heavily on the very specific structure of SLPs.

8.7 Favourable and unfavourable aspects of SLPs

In this section, we shall consider the favourable versus the unfavourable aspects of the SLP representation. With specific note of this representation in respect to the gcd function for polynomials.

8.7.1 Unfavourable aspects

- Deterministic equality is a very expensive operation. As noted before, the deterministic equality check, by current methods is exponential in the number of variables relative to evaluation of the SLP.
- Reduction operations e.g. quotient, remainder, gcd, lcm (least common multiple), etc. produce objects which will be larger than the original arguments. In comparison, the sparse form representation will form objects which may be smaller than the original arguments (given in sparse form). This may present garbage collection problems if one is using, for example, a Lisp based system.
- Because of the non-canonicality of the SLP representation it is very difficult for a human to recognise polynomials in this form. Because of this they must often be converted to a sparse form in order that they may be understood by a user.

8.7.2 Favourable aspects

- There are many objects which have much smaller assymptotic complexities, in the storage space required and the time taken to calculate them.
- There exist deterministic algorithms for calculating certain objects which are faster then the corresponding sparse form algorithms, e.g. derivation, calculation of determinants (for matrices of polynomials).
- Many non-deterministic algorithms exist for use on SLPs which in many cases are much faster than the corresponding algorithms which use the sparse form representation.

- There are a number of levels at which SLP algorithms may be effectively parallelised.

8.7.3 Specific conclusion for gcds

We now consider our specific case of the calculation of polynomial GCDs using SLP representation.
Our initial technique (that of using a Las Vegas method to get a result which was correct all the time) was rejected, as it made use of a deterministic equality check which became far to expensive as the number of variables increased.
We then proceeded to implement the Monte Carlo technique of section 6.5. We found that this was faster than the sparse representation based technique, provided that the degree of the polynomial was high enough and the number of variables was high enough. Other problems with the probabilistic technique was that it did not fit in with AXIOMs category structure, as detailed in 8.2.
For small polynomials the sparse form was better than both the Las Vegas and the Monte Carlo techniques.
Due to time restrictions, we have not been able to investigate fully the technique we proposed for the integration of modular SLPs into the AXIOM category structure. We leave this as a topic for possible further research. Other possible further research is the development of a Monte Carlo computer algebra system as detailed in section 8.4.

Appendix A

Aldor

Aldor is the new compiler language for the AXIOM computer algebra system. In order to obtain a full description of the ALDOR language we refer the reader to The Aldor User Guide [5].
Aldor attempts to achieve generality and power through simplicity. The strongest guiding directions for the language have been generality and uniformity, whilst retaining efficiency and portability.
Aldor is a strongly typed language, this means that within a specific *scope* a variable may only take values which are of one particular type. The compiler is very particular about the types of the parameters for any function, that is the parameters must have exactly the types that are specified by the signature (see section 2.2) of the function. AXIOM is built on top of Codemist Common Lisp (CCL, see appendix B) which is weakly typed, it is just this weakly typing of the Lisp, which allows a strongly typed system to be built over it, as the Lisp imposes no a-priori structure which might obstruct the structure required by Aldor, in turn the algebraic structure required by AXIOM.
Aldor can generate CCL (for AXIOM) or C for stand alone programs.
For the purposes of this thesis, we shall only describe a small subset of the language:

A.1 Domain syntax

The syntax of a Domain definition is as follows:
Domain_name(Domain_Parameter_List) : Domain_Category
== add_domain
Domain_name is the name associated with the domain, e.g. Integer.
Domain_Parameter_List is a list of parameters taken by the domain, these parameters may be objects or domains, the types must be given, in the case of parameters which are domains, the type will be a category, e.g. for the type `UnivariatePolynomial` the parameter list is `(x:Symbol,R:Ring)`, in this case `x` is the (major) variable and `R` is the Domain of its coefficients.
Domain_Category is the category of *Domain_name* it specifies the functions to be exported by *Domain_name.*
Domain_Category may take the form: *Predefined_Category* with *Additional_Exports*
Predefined_Category is some category which has already been defined,
Additional_Exports is a list of function signatures.

In explanation of the following example, we note that the AXIOM domain SemiGroup is intended to model the algebraic structure of a *semigroup.* A semigroup is a set (S) together with an operation, we shall denote by $*$, which is closed and associative on the set, that is,

$$\forall a, b \in S | a * b \in S$$

$$\forall a, b, c \in S | (a * b) * c = a * (b * c)$$

in the example $**$ and ^ denote 'raising to the power', or simply repeated application of $*$, viz.

$$a * *n = \underbrace{a * \cdots * a}_{n}$$

e.g. for SemiGroup the Category definition is:

```
SetCategory with {
      "*": (%,%) -> %;
      "**": (%,PositiveInteger) -> %;
      "^": (%,PositiveInteger) -> %}
```

This will export every function exported by SetCategory and also the functions "*","**","^" above.
`add_domain` contains all the implementation details. The syntax of `add_domain` is as follows: *Existent_Domain* add *function_implementation* Where *Existent_Domain* is some domain which has already been defined,
function_implementation contains code which implements the exported functions.

A.2 Imported Functions

Before using a function from another domain, it is necessary to import it. There are various ways to specify which functions to import:

- A function with signature **sig** from domain **Dom** may be specifically imported using the following syntax:

  ```
  import sig from Dom;
  ```

- If a variable is declared to be in a domain, then every function exported from that domain is imported.
- The domain of every parameter to a function and the return domain are imported, as in the point above.

function syntax: The general syntax for a function is:

```
fn(param_list):fn_type == {
  operation_list
}
```

where `fn` is the function name,
`param_list` is the parameter list together with types,
`fn_type` is the type of the value returned by the function,
`operation_list` is the body of the function, the last block must return a value which has the type `fn_Type`

assignment: to assign a value, `val` to a variable, `var`, we do the following:

```
var := val;
```

The basic arithmetic operations $(+, -, *, /)$ are denoted in the normal infix manner.

Appendix B

Codemist Common Lisp

The intermediate language used for the implementation of AXIOM, is Codemist Common Lisp (CCL). In order to obtain a full description of the CCL language we refer the reader to [6].
CCL is a *weakly typed* language, this means that variables are not required to be declared of a specific type, they may be assigned to any LISP object. LISP objects may be lists of other LISP objects, or atomic objects; examples of atomic objects are *strings, integers, symbols, etc.*
For the purposes of this thesis, we shall only describe a small subset of the language:

Function definition: in order to define a function called `fnName`, with a list of parameters `parameter_list`, and where the function is implemented by the operations in `operation_list`, we use the syntax:

```
(defun fnName (parameter_list)
  operation_list)
```

An anonymous function definition may be declared as follows, these are also known as *lambda expression.*

```
(lambda (parameter_list)
  operation_list)
```

We note that named functions support recursion, whereas anonymous functions do not.

array initialisation: in order to make an array object of length `array_length`, we may use the syntax:

```
(make-array array-length)
```

we should note that each element of the array will contain the element '`nil`'

array assignment: in order to assign an object, `object` to a variable `var_name`, we may use the syntax:

```
(setq var_name object)
```

element reference: to access the element at position `index` of array `array_name`, we may use the syntax:

```
(aref array_name index)
```

element assignment: to change the element at position `index` of array `array_name` to the value `value`, we may use the command '`setf`' together with the command '`aref`' in the following way:

```
(setf (aref array_name index) value)
```

We must note that the indexing starts at zero

remainder: in order to calculate the remainder of a value '`v`' with respect to some divisor '`d`', we may use the following command:

```
(rem v d)
```

The basic arithmetic operations (`+,-,*`) are denoted using postfix notation, e.g. to perform addition of two values `val1 and val2`, we couls use the code `(+ val1 val2)`

iterative constructs: CCL supports the *loop construct*, using the syntax:

```
(loop for i from base to top
  operation_list)
```

This will perform the commands in `operation_list` repeatedly, the index `i` taking the consecutive values between `base` and `top`

Appendix C

Chinese Remainder Theorem

C.1 A Modular Representation

In a modular representation a number is stored as a vector of residues. Each residue corresponds to a different modulus, each of which is relativley prime to all of the others. If the moduli are represented as $p_0, p_1, \cdots, p_{n-1}$ then an integer X would be represented as $\langle x_0, x_1, \cdots, x_{n-1} \rangle$ where $x_i \equiv X(\mathrm{mod} p_i)$, $0 \leq x_i < p_i$. The product of all the moduli will be called M and is equal to $p_0 p_1 \cdots p_{n-1}$.

C.2 The Chinese Remainder Theorem

The Chinese Remainder Theorem states that the system of simultaneous congruences,

$$\begin{aligned} X &\equiv x_0 \pmod{p_0} \\ X &\equiv x_1 \pmod{p_1} \\ &\vdots \\ X &\equiv x_{n-1} \pmod{p_{n-1}} \end{aligned} \tag{C.1}$$

has a unique solution mod $p_0 p_1 \cdots p_{n-1}$, assuming $\gcd(p_i, p_j) = 1$, for $0 \leq i, j < n$, $i \neq j$.

To show that any solution would be unique, we can consider two numbers A and B that obey the congruences of Equation C.1. Their difference $A - B$ must be divisible by all p_i, and hence M, so they are congruent mod$M$ as stated.

That a solution exists can be shown by considering the following equation

$$Y = \sum_{i=0}^{n-1} M_i y_i \tag{C.2}$$

$$\text{where } M_i = \frac{M}{p_i} \text{ and } y_i \equiv M_i^{-1} x_i \pmod{p_i}.$$

To show that it obeys each of the congruences of Equation C.1 we can consider the case of an arbitary p_i. As $M_j \equiv 0 \bmod p_i$ whenever $i \neq j$, then Equation C.2 can be reduced to the single term,

$$Y \equiv M_i M_i^{-1} x_i \equiv x_i \pmod{p_i}$$

so $Y \equiv X \bmod p_i$, for all p_i and hence $Y \equiv X \bmod M$, is a solution to the original set of congruences.

We shall consider how we may map other fields onto modular fields to allow us to make use of the CRT.

C.2.1 Integer to Mod

This conversion is easy if we have a bound to the size, in absolute value, of the integers we are considering, call the bound B.
To go from Integers to the modular field:
We shall work in a modular field $\mathbf{Z}_P$ where $P > 2B$,
denote the integer to be represented as x, denote the modular value which will represent x as m, also denote $\lfloor (P/2) \rfloor$ as M.

To go from Integers to the modular field:

if $x < 0$ then $m \leftarrow P + x$, otherwise $m \leftarrow x$.

To go from the modular field to the Integers:
if $m > M$ then $x \leftarrow m - P$ else $x \leftarrow m$

C.2.2 Rationals to Mod

For this conversion, we use a technique taken from [24]. We need a bound to the size of the denominator and to the size of the numerator of the rational we are considering, we call the bound B.
We shall work in a modular field $\mathbf{Z}_P$ where $B \leq \sqrt{P/2}$,
We denote the rational to be represented as x also the modular value which will represent x as m.

To go from the rationals to the modular field:
if $x = v_1/v_2$, where $v_1, v_2 \in \mathbf{Z}$
then we use Euclids algorithm over $\mathbf{Z}$ to find the value $v_2^{-1} \in \mathbf{Z}_P$, now we may calculate the product $v_1 v_2^{-1}$ in the field $\mathbf{Z}_P$ to arrive at the representation required.

To go from the modular field to the rationals:
We use the following algorithm:

1) $u \leftarrow (1, 0, P)$, $v \leftarrow (0, 1, m)$

2) While $\sqrt{P/2} \leq v.3$ do

2.2) $\{q \leftarrow \lfloor (u.3)/(v.3) \rfloor$, $r \leftarrow u - qv$, $u \leftarrow v$, $v \leftarrow r\}$

4) if $|(v.2)| \geq \sqrt{m/2}$ then error()

5) Return (v.3,v.2)

now the rational value we require is $\frac{v.3}{v.2}$
We should note that the condition in step 4) will only be satisfied if the field $\mathbf{Z}_P$ is not big enough, or a modular value has been chosen, which does not correspond to a rational. So long as our SLP algorithms are correct, we should not encounter this problem.

Appendix D

Bareiss method

Following the method described in [1] we introduce the notation

$$a_{ij}^{(k)} = \begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1k} & \cdots & a_{1j} \\ a_{21} & a_{22} & \cdots & a_{2k} & \cdots & a_{2j} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{k1} & a_{k2} & \cdots & a_{kk} & \cdots & a_{kj} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{i1} & a_{i2} & \cdots & a_{ik} & \cdots & a_{ij} \end{vmatrix} \quad (k < i, j \leq n)$$

We also recall the identities that Bareiss proves:

$$a_{ij}^{(k)} = \frac{1}{a_{k-1,k-1}^{(k-2)}} \begin{vmatrix} a_{kk}^{(k-1)} & a_{kj}^{(k-1)} \\ a_{ik}^{(k-1)} & a_{ij}^{(k-1)} \end{vmatrix} \tag{D.1}$$

with the conditions that:

$$a_{00}^{-1} = 1 \quad a_{ij}^{0} = a_{ij} \ (i, j = 1, \cdots, n)$$

Using these identities we may yield succesive principal minors $a_{k+1,k+1}^{(k)}$ and thus lead to an efficient calculation of the determinant $|A| = a_{n,n}^{(n-1)}$
We notice that using this method to form an SLP which represents the determi-

nant of a three by three symbolic matrix:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

Involves eleven times operations, one divide and five subtractions. We see that in fact due to cancellation and factorisation, this may in fact be represented in the following optimum form:

$$a_{11}(a_{22}a_{33} - a_{23}a_{32}) - a_{31}(a_{22}a_{13} - a_{23}a_{12}) - a_{21}(a_{12}a_{33} - a_{13}a_{32}) \tag{D.2}$$

This could be translated into an optimal SLP containing nine times operations and five minus operations, a saving of two times and one divide operations. We may in fact use this in a more general setting, Bareiss gives a more general identity:

$$a_{ij}^{(k)} = \frac{1}{[a_{ll}^{(l-1)}]^{k-l}} \begin{vmatrix} a_{l+1,l+1}^{(l)} & \cdots & a_{k+1,k}^{(l)} & a_{l+1,j}^{(l)} \\ \cdots & \cdots & \cdots & \cdots \\ a_{k,l+1}^{(l)} & \cdots & a_{k,k}^{(l)} & a_{k,j}^{(l)} \\ a_{i,l+1}^{(l)} & \cdots & a_{i,k}^{(l)} & a_{i,j}^{(l)} \end{vmatrix} \tag{D.3}$$

we may take l=k-2 and so this reduces to:

$$a_{ij}^{(k)} = \frac{1}{[a_{k-2,k-2}^{(k-3)}]^2} \begin{vmatrix} a_{k-1,k-1}^{(k-2)} & a_{k-1,k}^{(k-2)} & a_{k-1,j}^{(k-2)} \\ a_{k,k-1}^{(k-2)} & a_{k,k}^{(k-2)} & a_{k,j}^{(k-2)} \\ a_{i,k-1}^{(k-2)} & a_{i,k}^{(k-2)} & a_{i,j}^{(k-2)} \end{vmatrix} \tag{D.4}$$

so long as $k > 2$ we could use this identity along with the form given in D.2 terminating the recursion by using identity D.1.

References

[1] E. H. Bareiss. Sylvesters identity and multistep integer preserving gaussian elimination. *Mathematics of Computation*, 22:565–578, 1968.

[2] S. J. Berkowitz. On computing the determinant in small parallel time using a small number of processors. *Information Processing Letters*, 18:147–150, 1984.

[3] A. Díaz and Kaltofen E. Foxbox: A system for manipulating symbolic objects in black box representation. In *ISSAC 98, University of Rostock*, 1998.

[4] M. F. Singer E. Kaltofen. Size efficient parallel algebraic circuits for partial derivatives. In *IV International Conference on Computer Algebra in Physical Research*, pages 133–145, 1991.

[5] Watt S.M. et al. *AXIOM Library Compiler User Guide.* The Numerical Algorithms Group Limited, 1995.

[6] Steele G.L. http://kmi.open.ac.uk/cltl/cltl2.html, 1990.

[7] O.H. Ibarra, S Moran, and L.E. Rosier. Probabilistic algorithms and straight-line programs for some rank decision problems. *Information Processing Letters*, 1981.

[8] R. D. Jenks. A language for computational algebra. *Proceedings of the 1981 ACM Symposium on Symbolic and Algebraic Computation*, pages 6–13, 1981.

[9] Y.Siret J.H.Davenport and E.Tournier. *Computer Algebra Systems and Algorithms for Algebraic Computation.* Academic Press Ltd., 1993.

[10] J.T.Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the Association for Computing Machinery*, 27(4):701–717, October 1980.

[11] E. Kaltofen. Greatest common divisors of polynomials given by straight-line programs. *Journal of the ACM*, 35:231–264, 1987.

[12] E. Kaltofen. Gröbenor basis and polynomial ideal theory. 1996.

[13] E. Kaltofen and B.M. Trager. Computing with polynomials given by black boxes for their evaluations: Greatest common divisors, factorisation, seperation of numerators and denominators. *Journal of Symbolic Computation*, 9(3):301–320, 1990.

[14] D. Knuth. *Semi-Numerical Algorithms, The Art of Computer Programming*, volume 2. Addison Wesley, 1980.

[15] J. D. Lipson. *Elements of Algebra and Algebraic Computing.* Addison-Wesley Publishing Company, 1981.

[16] R. Loos. Generalised polynomial remainder sequencess. In *Computer Algebra, Symbolic and Algebraic Computation*, pages 115–137. Springer-Verlag, second edition, 1983.

[17] Guillermo Matera. Integration of multivariate rational functions given by straight-line programs. In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, Departmento de Matemáicas, Fac. de Ciancias Exactas, Univ. de Buenos Aires., 1995.

[18] M. Mignotte. *Mathematics for Computer Algebra.* Springer-Verlag, 1992.

[19] G.L. Miller, V. Ramachandran, and E. Kaltofen. Efficient parallel evaluation of straight-line code and arithmetic circuits. *SIAM Journal of Computing*, 17:687–695, 1988.

[20] R.T. Moenk. Fast computation of gcds. *Proceedings of 5'th ACM Symposium on the Theory of Computation*, pages 142–151, 1973.

[21] Doye N.J. *Order Sorted Computer Algebra and Coercions.* PhD thesis, University of Bath, 1997.

[22] L.M. Pardo. How lower and upper complexity bounds meet in elimination theory. In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, 1995.

[23] O. Pretzel. *Error Correcting Codes and Finite Fields.* Clarendon Press, Oxford, 1996.

[24] J.H.Davenport P.S.Wang, M.J.T.Guy. P-adic reconstruction of rational numbers. *ACM SIGSAM Bulletin*, 16:2–3, 1982.

[25] S Puddu and J Sabia. An effective algorithm for quantifier elimination over algebraically closed fields using straight-line programs. 1995.

[26] N. Revol. *Complexité de l'évaluation parallèle de circuits arithmétiques.* PhD thesis, l'Institut National Polytechnique de Grenoble, 1992.

[27] Roch J.-L. Revol N. Parallel evaluation of arithmetic circuits. Preprint submitted to Elsevier Preprint, 1996.

[28] K. O. Geddes S. R. Czapor. A comparison of algorithms for the symbolic computation of padé approximants. In *EUROSAM 84*, volume 174 of *Lecture Notes in Computer Science*, pages 248–259. Springer-Verlag, 1984.

[29] M. Shub & S. Smale. Computational complexity - on the geometry of polynomials and a theory of cost, part i. *NA*, NA, 1982.

[30] R. Zippel. *Effective Polynomial Computation.* Kluwer Academic Publishers, 1993.