Gareth Taylor
glt1000

Solving Certain p-adic Integrals

Diploma in Computer Science

Corpus Christi College

2002

Proforma

Name:	Gareth Taylor
College:	Corpus Christi
Project Title:	Solving Certain p-adic Integrals
Examination:	Diploma in Computer Science
Year:	2002
Approximate Word-Count:	11200
Project Originator:	Gareth Taylor
Project Supervisor:	Ben Harris

Original Aims

The aim was to create a system which, given a p-adic integral of a certain form, could solve the integral and produce output in a format suitable for an algebra package such as Maple.

Work Completed

A program to solve a restricted form of the integrals was created successfully. A parser was written to enable easy input of integrals, also allowing for them to occur amidst other text, such as Maple commands. However, independence of Maple was desired, and appropriate symbolic algebra facilities were added to achieve this. This also produced a speed increase over Maple, through knowing in advance the approximate form the results should take.

Special Difficulties

None.

Declaration of Originality

I, Gareth Taylor of Corpus Christi College, being a candidate for the Diploma in Computer Science, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed   Date

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Background Mathematics 2

1.3 Previous Work 4

2 Preparation **5**

2.1 Project Design 5

2.2 Considerations and Requirements 6

2.3 Backups 8

3 Implementation **9**

3.1 Solver 10

3.2 Parser 18

3.3 Simplifier 22

3.4 Trees 29

4 Evaluation **33**

4.1 Parser 33

4.2 Solver 35

4.3 Simplifier 36

4.4 General integrals 37

4.5 Efficiency 39

5 Conclusions **41**

5.1 Improvements 41

5.2 Uses and possibilities 42

Bibliography **45**

A Lie Algebras and Zeta Functions **47**

B The Heisenberg Lie Algebra **51**

C A Full Example **53**

Project Proposal **59**

Chapter 1

Introduction

Mathematical packages for performing integration have existed for a long time, taking a variety of approaches: producing approximate results through numerical methods, or exact formulae by symbolic algebra. The integrals involved are traditionally over the real or complex numbers, whose properties allow the numerical analysis to work, but can also prevent exact calculation.

This project is concerned with integrals of a different nature, for which there do not exist such packages. These involve p-adic numbers, and their method of solution falls somewhere between the approximate and exact approaches. The aim is to produce a system which will solve p-adic integrals of a certain form.

1.1 Motivation

The roots of the project lie in group theory and in recent work on methods for calculating rates of growth of subgroups of groups, or subalgebras of Lie algebras. In a paper by Grunewald, Segal and Smith [4], the rate of growth is encoded as a sum, called a *zeta function*, and it is shown that this sum is expressible as a p-adic integral.

My recent Ph.D. was concerned with solving these p-adic integrals, and the process would involve a combination of cunning geometric manipulations and painstaking direct integral solving. This would have been made much faster if the later parts, where the 'cunning' had been used up, had been automated.

Accordingly, this project is designed to achieve this. It is unfortunately too late to help with my Ph.D., but should be useful to others in the field. Indeed, my Ph.D. supervisor, Marcus du Sautoy, has expressed an interest in it, and has suggested features that should be included.

1.2 Background Mathematics

While the background theory is rather complicated, the majority of it is not required to understand the project. Appendix A contains an informal description of where these integrals come from and why they are worth solving. The maths below is more directly relevant, and includes a demonstration of some of the methods employed. It also shows some of the information that an integral solver would need to know, and some of the procedures it would have to be able to apply.

1.2.1 p-adic Numbers

The first thing to note is that these are not 'ordinary' integrals over the real numbers. The integrals here will be using p-adic numbers instead, and these are described below.

Given a prime number p, any $n \in \mathbb{N}$ may be written as $n = p^m r$, where $m, r \in \mathbb{N}$ and r is coprime to p. (Note that $\mathbb{N}$ is taken to include 0.) Let $v(n) = m$, and define the *p-adic valuation* of n to be $|n| = p^{-v(n)} = p^{-m}$.

Any $n \in \mathbb{N}$ may be written in base p as a sum $\sum a_i p^i$, with $0 \leq a_i < p$, in which case $v(n)$ would equal the index of the smallest non-zero coefficient a_i. While ordinary integers give finite sums of this form, the definition of the p-adic valuation means that infinite sums will also converge with respect to the valuation. This gives an extension to the integers – the *p-adic integers*, written $\mathbb{Z}_p$.

Manipulating these sums as formal power series shows that if $a, b \in \mathbb{Z}_p$, then $v(ab) = v(a) + v(b)$, and that a divides b if and only if $v(a) \leq v(b)$, with $v(b/a) = v(b) - v(a)$. Sums are less easy to deal with: for example, if $a = 1$ and $b = p^k - 1$, for some k, then $v(a) = v(b) = 0$, but $v(a + b) = k$. This may be dealt with using the mathematical tricks mentioned earlier, but will not be relevant here.

Notice that the leading coefficient itself doesn't contribute – only its index determines the valuation. And just as a piecewise constant function over the reals may be integrated by considering the regions in which it is constant, a p-adic integral of a function of valuations may be considered by looking at a single representative for a given valuation, and multiplying by the size of the set of p-adic integers that have that valuation. To find the size of subsets of $\mathbb{Z}_p$, the *Haar measure*[1] on $\mathbb{Z}_p$ is used, which is defined so that $\nu(\mathbb{Z}_p) = 1$ and, in general, so that $\nu(p^n \mathbb{Z}_p) = p^{-n}$, for $n \in \mathbb{N}$.

1.2.2 Cone Integrals

Define a *cone integral* to have a certain form, as follows: for a fixed prime p, and polynomials $f_0, g_0, \ldots, f_k, g_k$ in d variables and with integer coefficients, the associated cone integral is

$$I(s) = \int_{V_p} |f_0(\mathbf{x})|^s |g_0(\mathbf{x})|^{-1} d\nu,$$

where $V_p = \{\mathbf{x} \in \mathbb{Z}_p^d : v(f_i(\mathbf{x})) \leq v(g_i(\mathbf{x})) \text{ for } i = 1, \ldots, k\}$.

To save space, the expression "$v(f_i(\mathbf{x})) \leq v(g_i(\mathbf{x}))$" shall be replaced by the equivalent but more concise "$f_i(\mathbf{x}) \mid g_i(\mathbf{x})$".

An example. To illustrate, here is a very simple example in detail. Let

$$I = \int_{x \in \mathbb{Z}_p} |x|^{s-1} d\nu.$$

[1] Note that this is denoted by the Greek letter ν, not to be confused with the valuations, denoted by v.

Currently, x ranges over all possible valuations. The integral is broken into a sum of the possible valuations x can take – x has valuation exactly m if and only if $x \in p^m\mathbb{Z}_p \setminus p^{m+1}\mathbb{Z}_p$, a subset of $\mathbb{Z}_p$ with Haar measure $\nu(p^m\mathbb{Z}_p) - \nu(p^{m+1}\mathbb{Z}_p) = p^{-m} - p^{-(m+1)}$. Thus,

$$I = \sum_{m\in\mathbb{N}} \int_{x\in p^m\mathbb{Z}_p\setminus p^{m+1}\mathbb{Z}_p} |x|^{s-1} d\nu.$$

Each sub-integral is over x of constant valuation, and so may be integrated as (function value)$\times$(measure of set), which gives

$$I = \sum_{m\in\mathbb{N}} p^{-m(s-1)}(p^{-m} - p^{-m-1}) = (1-p^{-1})\sum_{m\in\mathbb{N}} p^{-ms} = \frac{1-p^{-1}}{1-p^{-s}}.$$

The methods for dealing with integrals with conditions will be covered in Chapter 3, where it is more appropriate.

1.3 Previous Work

Much of the work in this area has been mathematical rather than computational. Zeta functions of groups have been popular recently, and test cases are taken later from a number of papers in the area. However, there do exist some relevant computing resources. There are a number of algebra packages, such as Maple[7] and Mathematica[11]. These are useful for the adding together of final results rather than the integrating process. Another package is Magma[6], which appears to do more sophisticated mathematics[2], and may be able to assist with the algebraic geometry part of the integration.

Some of the 'tricks' mentioned above have been automated[1], however I was unable to find any existing programs for solving p-adic integrals, so this project will be based upon the mathematical theory only.

[2] I apologise to the others if I am doing them a disservice – I am basing the comments on webpage descriptions and only limited experience.

Chapter 2

Preparation

2.1 Project Design

This project originated from an idea of my own, and so before even the proposal could be written, it was necessary to consider the details of the project and come up with a set of goals that could be achieved in the available time.

One of the first decisions that had to be made was to place some sort of limit on the integrals the project would be required to solve. As with 'ordinary' real integrals, p-adic integrals can get arbitrarily complicated, and it is only recently that methods have been developed to solve some of the simpler forms one encounters. A sensible first restriction was to consider only the sorts of integral that were described in the Introduction – those resulting from the calculation of zeta functions of algebras – rather than trying for general p-adic integrals.

Another decision involved the output – what would the program be expected to produce? There already exist procedures to find approximate results by numerical methods, but one advantage of the p-adic integrals to be considered is that they may gradually be broken down into smaller pieces, which may eventually be summed to give an exact answer as a function of the prime p. This final summation is always of rational functions, an ideal task for symbolic algebra packages such as Maple. Therefore, an acceptable

output would be a series of sums expressed as valid input to such a package.

This was the essence of the final project proposal – to take a restricted form of p-adic integral, break it up appropriately and solve each sub-integral, then produce the final solution as an expression acceptable to Maple.

2.1.1 Proposal Refinements

The first period of planning which followed the proposal submission suggested some changes to the aims of the project. It had been written shortly after the completion of my Ph.D., and that had encouraged more of a mathematical than a computing project plan. I decided that a more appropriate aim would be to consider only integrals whose conditions are of the form *monomial divides monomial*, rather than the original *monomial divides homogeneous polynomial* form.

This may sound a substantial reduction, but in general one may use various mathematical tricks to reduce the polynomial cases to sums of monomial ones. Such tricks fall more within the realm of algebraic geometry, and so, with the agreement of my supervisor, I decided not to involve them and to deal instead with just the monomial examples.

2.2 Considerations and Requirements

Before beginning any programming, there was a period of planning, reading and discussion, concerning choices that would affect the development of the project. Having just written a thesis in the subject may count as an amount of preparation for this project, but much of that was more mathematical than is being considered here. However, one of the first tasks was to read through both what I had written and papers by others, to refresh myself and to check on the common formats and methods people use.

2.2.1 Java vs. ML

At the time of the project proposal, my recent programming experience had been with Java, and I felt that it would be a reasonable choice for the project. However, my supervisor suggested that I consider ML, since the integrals that

would be considered have a somewhat recursive method of solution. I spent a while reading and learning ML, since this was before the Lent Term course on the language, but in the end decided that Java was more suitable. ML, while being good at recursion, seemed a little too restrictive in other areas – such as manipulating strings for input and output, and for user interaction throughout.

2.2.2 Parsing Input

A program which solves integrals would need some way of recognising an integral in the first place. This would require deciding upon some input format for an integral, and then writing a parser to interpret it. Fortunately, the form in which most are written is fairly standard, with the main variation being the use of "$v(f) \leq v(g)$" or "$f \mid g$". These are equivalent, as described earlier, and the second form was chosen.

Having a sensible method for parsing an integral would allow for more than just a solver – it could act as a form of preprocessing to Maple, or other such packages. For example, the input file could contain integrals amidst Maple commands, such as

`simplify(`*integral to be solved*`);`

This would be converted to

`simplify(`*solved integral in Maple format*`);`

This would be very useful, such as when an integral with polynomial conditions is broken into a sum of monomial ones, as these could then be summed easily.

The Lent Term course on Compiler Construction contains details about writing parsers, and so it was decided to leave programming this part until after that course.

2.2.3 Extra Features

Discussion with my Ph.D. supervisor, Marcus du Sautoy, produced some more good ideas. Being more fond of Mathematica, Marcus felt that rather

than being constrained to Maple, the output should be in a more general format, with a conversion to Maple or other formats at the end. This is certainly sensible, and was included as part of the plan.

We also decided that as well as being provided with a fully automated integral solver, the user should be given the ability to choose the paths the solution takes (by suggesting *blow-ups*, described in Chapter 3). Of course, the solver should work through by itself if the user doesn't wish to join in.

2.2.4 Test Data

A number of test integrals were taken from my thesis [9], and more from other recent papers in the area [3], [4], [10].

The largest and most complicated integral I worked out by hand, the unpleasant-looking

$$\int_{\substack{y \mid ew \\ y \mid uwx^2 \\ z \mid xy \\ y \mid uxa \\ z \mid x^2c \\ y \mid exa}} |u|^{2s-5}|w|^{2s-4}|x|^{2s-2}|y|^{s-4}|z|^{s-4}d\nu$$

was used frequently as a test case for efficiency. By hand, this took several days to solve, and I hoped that the project might do a little better!

2.3 Backups

Code for the project was written on the University's Central Unix Service machines (CUS), and back-up copies were taken regularly – usually daily on the Computing Service's machine Thor when actively writing, and weekly on a private machine.

The write-up was written on said private machine, and backed-up often to CUS.

Chapter 3

Implementation

The final structure of the project consists of three main modules:

Parser. This takes an input file and extracts from it integrals that are to be solved, converting into an appropriate internal format. Any surrounding text in the input file is left, as it may be used later – such as the 'simplify' example earlier.

Solver. This takes the interpreted version of the integral and performs a series of transformations, gradually solving the integral by breaking it into smaller problems. This originally produced output in the form of a string valid for Maple, but the final version produces a tree structure.

Simplifier. This performs a series of summations and factorisations of the results of the Solver, producing a single result rather than a sum of many. This means that the program no longer requires Maple to find a final solution, and does so at least as fast as Maple.

These will be discussed in the order they were written rather than that above, as that will explain choices made as the project progressed. That order is: the original Solver, the Parser, the Simplifier and finally the revisions using Trees.

3.1 Solver

The first task was to create a format suitable for storing and manipulating the integrals. Recall the form of the integrals involved:

$$I(s) = \int_{f_i(\mathbf{x}) \mid g_i(\mathbf{x}) \ (i=1,\dots k)} |f_0(\mathbf{x})|^s |g_0(\mathbf{x})|^{-1} d\nu.$$

An integral of this form is determined by $2k+2$ monomial terms: two for the integrand and k pairs forming conditions. The names of the variables are not significant, providing they remain distinct, and so a monomial is determined solely by a vector of integer indices, indictating the powers of each of the variables occurring in a monomial.

As an example, consider the *Heisenberg Lie Algebra.* which shall be used for demonstration purposes throughout. It gives rise to the p-adic integral

$$I = \int_{z|xy} |x|^{s-1}|y|^{s-2}|z|^{s-3} d\nu.$$

Using triplets to encode the indices of x, y and z, the four monomials here may be represented as:

$$\begin{array}{lcl} f_0 & : & \{1,1,1\} \\ g_0 & : & \{1,2,3\} \\ f_1 & : & \{0,0,1\} \\ g_1 & : & \{1,1,0\}, \end{array}$$

and thus the entire integral may be encoded by the fairly short string "`{1,1,1}{1,2,3}{0,0,1}{1,1,0}`". This was deemed a sufficient method for input at this stage, to be replaced by a decent integral parser at a later stage.

Such information could be stored internally as a two-dimensional `int[][]` array, but this would result in fairly unintelligible code and impose inconvenient restrictions, such as monomials having to be the same length.

A sensible way of storing this information internally is to use simple classes for the various parts, and the following were created: a class *Monomial* which is essentially be an integer array encoding the indices, and a class *Integral* containing an array of Monomials among its data fields.[1] This allows some freedom: for example, monomials of different lengths may be stored easily, and g_0 above could be truncated to just $\{1, 1\}$.

Note. The mathematical notation convention is retained. This means that the integrand terms have the subscript 0 and the conditions count from 1 upwards. It would be more traditional programming to label the integrands just f and g, say, and have the conditions named $f_0, \ldots, g_k$. Since much of this information is passed around when creating sub-integrals, keeping everything together seemed convenient, as well as avoiding confusion for anyone who is used to working with the integrals mathematically.

This format enabled the writing of a series of methods to simplify a integral, before attempting to solve it. Such simplifications include the straightforward:

- *cancelling variables.* If a variable occurs on both sides of a condition, then it may simply be cancelled from each. For example, $ax \mid ayz$ reduces to $x \mid yz$.
- *removing irrelevant conditions.* If some f_i $(i > 0)$ is stored as $\{0, \ldots, 0\}$, then it represents 1, and any condition of the form $1 \mid g_i$ should be removed, as it says nothing useful.[2]

and the slightly deeper:

- *removing units.* If some g_i $(i > 0)$ is stored as $\{0, \ldots, 0\}$ and thus is 1, then the condition $f_i \mid 1$ must mean that every variable appearing in f_i

[1] Using a series of Java's lists would have allowed for more length flexibility than arrays, but would have introduced more features than were required, so the Monomials kept to the simplicity of integer arrays.

[2] Strictly, $\{0, \ldots, 0\}$ represents some *unit* in $\mathbb{Z}_p$, that being an element of valuation 0. But it may be assumed to be 1, since everything is multiplicative.

must have valuation 0. As everything here is multiplicative, all occurrences of such variables, including in the integrand, may be removed, as may the condition.

- *removing redundant conditions.* Suppose there are the two conditions $a \mid by$ and $ax \mid b$. The second implies that $a \mid b$, so the first is redundant and may be removed. In general, if $f_i \mid f_j$ and $g_j \mid g_i$ then condition i is redundant.

Other methods written to handle Monomials and Integrals consist of some management procedures, such as

- *truncating monomials.* As mentioned in the example above, storing {`1,1`} keeps as much information as {`1,1,0`}. Monomials are truncated on input where appropriate, to save time when searching through them.

- *"deep cloning".* When breaking an integral into smaller parts (see the next section), it would take some effort to construct it from nothing each time. It is much easier to take a copy of the parent integral and make the appropriate changes. The cloning methods[3] copy an integral and its monomial information at the integer level, rather than just creating more references to the same information. This allows manipulation of each child integral independently, without altering the information stored in the others.

- *displaying integrals.* While an integral may be input (for now) as a string of arrays, it is not very nice to read. Accordingly, there is a *toString* method to print it so that it looks like an actual integral. This consists of allocating variables $a, b, \ldots$ to the entries in the arrays and interpreting the elements as exponents, adding valuation signs or a $d\nu$ where appropriate.

3.1.1 Blow-ups

In an ordinary integral, it is common to perform a change of variables to make the problem easier; with p-adic integrals, however, variable changes

[3] Called *deepClone()* to distinguish them from the usual shallow *clone()*.

cannot be made quite so freely. As described in Section 1.2.1, summing p-adic integers often loses control of valuations, and division is only permitted if the result is still a p-adic integer.[4]

Suppose a problem involves the variables x and y, and that dividing one by the other would make the problem easier. For example, it may involve the condition $ax \mid by$. Substituting $y' = y/x$, would change this to $a \mid by'$. Similarly, letting $x' = x/y$ would produce $ax' \mid b$. Both of these are simpler conditions, but there is no guarantee that either transformation is valid, since x and y will be varying over a range of valuations.

To solve this problem, a transformation called a *blow-up* is used. This is one of the 'tricks' mentioned earlier on, but may be used here in a simple way here to deal with the above situation. The result is to split the integral into regions, or *branches*: $v(x) \leq v(y)$ and $v(x) > v(y)$. The changes of variables $y' = y/x$ and $x' = x/y$ are then valid on the two parts respectively, as the valuation constraints guarantee that the results are p-adic integers.

It turned out, when writing the code, that it was easier for reasons of symmetry to break into three branches: $v(x) \leq v(y)$, $v(x) \geq v(y)$ and $v(x) = v(y)$. Denoting the sub-integrals by I_1, I_2, I_3 gives $I = I_1 + I_2 - I_3$.

The transformation $y' = y/x$ is equivalent to replacing every occurrence of the variable y by the product xy'. Using the array notation for monomials, this involves incrementing the x-element of a monomial's integer array by the number in the y-element place. The transformation $x' = x/y$ is of course similar.

For the region $v(x) = v(y)$, either of x or y may be divided by the other, so let $y' = y/x$ again. The arrays are altered as before, but the new variable y' has valuation zero. This can be recorded by adding the extra condition $y' \mid 1$. This will be caught be the "remove units" method on the next pass through the integrals.

The use of y' does not require extending the arrays of monomials, since there is still the same number of variables: y has been removed, y' has been added. So y' may take the place of y in the arrays.

[4] Recall $v(y/x) = v(y) - v(x)$. This must be non-negative.

Following a blow-up, the original integral has been broken into smaller and simpler integrals, and new Integral objects are created and solved recursively. This will continue until integrals with no remaining conditions are reached. Such a state will be reached if the choices of blow-up variables are sensible: choosing both to be on the same side of a condition, say from the same f_i, will not help, for this will make that monomial larger without any increased cancelling possibilities. Choosing a variable from either side of the same condition is better, as this will guarantee some cancelling.

To help the decision process, the user may enter a pair of variables to use for a blow-up. If, however, they decide that the program should take over, the computer can finish that branch of the blow-up. The user will be asked again upon entry to the next branch at which they last interacted.

When the program is choosing variables for a blow-up, it proceeds through the conditions in order, picking a variable from each side and gradually reducing a condition before progressing on to the next. To speed the solving process, the variables chosen are those occurring with maximal exponent on each side, as this will give the most cancelling of terms.

Based on blow-ups, there is the following further simplifying method:

- *monomial divides variable.* Suppose there is the condition $xyz \mid a$. The transformation $a' = a/xyz$ is valid, since a' must be a p-adic integer. The other parts of the blow-ups will not occur.[5] This would be done in the same way as a blow-up: incrementing x, y, z by occurrences of a.

Example. Recall the Heisenberg integral from earlier:

$$I = \int_{z|xy} |x|^{s-1}|y|^{s-2}|z|^{s-3}d\nu.$$

[5] More precisely, they will occur, but will not contribute. Take the condition $x|y$. The blow-up part with $v(x) \leq v(y)$ makes no change; that with $v(x) \geq v(y)$ reduces to equality; the third part, $v(x) = v(y)$, cancels this off when subtracted. These last two parts may therefore be ignored.

A suitable pair of variables for a blow-up is x and z. (Choosing y and z is as valid, but choosing x and y will not help.) This means that the integral is broken into three parts, making the appropriate changes of variables on each part.

Consider the part given by $v(z) \leq v(x)$. The change of variables means that x is replaced by $x'z$, so the integral becomes

$$I_1 = \int_{z|x'zy} |x'z|^{s-1}|y|^{s-2}|z|^{s-3}|z| \; d\nu = \int |x'|^{s-1}|y|^{s-2}|z|^{2s-3} \; d\nu,$$

where the extra $|z|$ before the $d\nu$ is the Jacobian of the transformation. The z terms would now cancel to give the condition $1 \mid x'y$, which would then be removed completely for being irrelevant.

In the array notation, the following has occurred.

Original integral: `{1,1,1}{1,2,3}{0,0,1}{1,1,0}`.

The blow-up, with $x \to x'z$, means that each z element is incremented by the corresponding x. To take account of the extra $|z|$ factor of the Jacobian, a further 1 is subtracted from the z element of g_0.

Blown-up integral: `{1,1,2}{1,2,3}{0,0,1}{1,1,1}`.

↑ *the changes happen to cancel out here.*

Cancel z terms: `{1,1,2}{1,2,3}{0,0,0}{1,1,0}`

Remove the irrelevant condtion: `{1,1,2}{1,2,3}`

The other branches are similar.

3.1.2 Condition-free integrals

A simple integral with no conditions was solved as an example, back in Section 1.2.2. In general, a similar calculation shows that

$$\int |x|^{as-b-1} d\nu = \frac{1-p^{-1}}{1-p^{-(as-b)}}.$$

Rather than performing the geometric progression each time, this is simply noted in the program, and a condition-free integral is converted directly into a string in Maple encoding the result. In later versions of the Solver, this method of output was changed, but is discussed here to preserve chronology.

Each remaining variable in the integrand is replaced by the string `(1-X^a*Y^b)`, where a, b are the appropriate indices.[6] The output is tidied in the cases where X or Y equal 0 or 1, by ignoring the unnecessary exponents or terms.[7] These are then concatenated together, adding a "`*`" between consecutive terms. Finally, the output is enclosed between "`1/(`" and "`)`". This is thus a valid expression for a term in Maple.

This string is then passed back through the series of recursive calls, being joined together with "`+`", "`-`" or more brackets as appropriate, producing the final output of a large string of Maple. The result is enclosed in the command "`simplify( ... );`", so that Maple returns does not just return the string it was given without alteration.

Example. The Heisenberg integral once again. The first branch of the blow-up had the sub-integral

$$I_1 = \int |x'|^{s-1}|y|^{s-2}|z|^{2s-3} d\nu \implies \{1,1,2\}\{1,2,3\},$$

which would produce the output `1/((1-X)*(1-X*Y)*(1-X^2*Y^2))`.

[6]It is traditional when writing these functions to use the substitutions $X = p^{-s}, Y = p$.

[7]Also, for simplicity, the $(1-p^{-1})$ factors are ignored in the output. These always occur by the nature of the integrals and are cancelled off when used, as in the expression for $\zeta_{G,p}(s)$ in appendix A. In effect, there is an implicit $(1-p^{-1})^d$ factor for an integral with d variables in the integrand.

The other branches of the main integral are as follows:

- $v(z) \geq v(x)$. Replace z by xz', giving

$$I_2 = \int\limits_{xz'|xy} |x|^{s-1}|y|^{s-2}|xz'|^{s-3}|x|d\nu = \int\limits_{z'|y} |x|^{2s-3}|y|^{s-2}|z'|^{s-3}d\nu.$$

Since $z|y$, the transformation $y \to y'z$ is valid, giving

$$I_2 = \int\limits_{z'|y'z'} |x|^{2s-3}|y'z'|^{s-2}|z'|^{s-3}|z'|d\nu = \int |x|^{2s-3}|y'|^{s-2}|z'|^{2s-4}d\nu,$$

The sequence of array transforms here is:

Original	:	{1, 1, 1}{1, 2, 3}{0, 0, 1}{1, 1, 0}
$z \to xz'$	:	{2, 1, 1}{3, 2, 3}{1, 0, 1}{1, 1, 0}
Cancel x terms	:	{2, 1, 1}{3, 2, 3}{0, 0, 1}{0, 1, 0}
$y \to y'z'$	:	{2, 1, 2}{3, 2, 4}{0, 0, 1}{0, 1, 1}
Cancel z' terms	:	{2, 1, 1}{3, 2, 3}{0, 0, 0}{0, 1, 0}
Remove irrelevant	:	{2, 1, 2}{3, 2, 4}

which gives the output `1/((1-X^2*Y^2)*(1-X*Y)*(1-X^2*Y^3))`.

- $v(z) = v(x)$. Replace z by xz', and add the condition $z'|1$, thus:

$$I_3 = \int\limits_{\substack{z'|y \\ z'|1}} |x|^{2s-3}|y|^{s-2}|z'|^{s-3}d\nu = \int |x|^{2s-3}|y|^{s-2}d\nu,$$

where the '$z'|1$' condition was caught by the *remove units* method, making the other condition into '$1|y$', which was removed for being irrelevant. This gives the output expression `1/((1-X^2*Y^2)*(1-X*Y))`.

Therefore, performing *solve(I)* returns

simplify(*solve*(I_1) + *solve*(I_2) - *solve*(I_3));

which recurses down to give the output string:

```
simplify(1/((1-X)*(1-X*Y)*(1-X^2*Y^2))
        +1/((1-X^2*Y^2)*(1-X*Y)*(1-X^2*Y^3))
        -1/((1-X^2*Y^2)*(1-X*Y)));
```

This may be passed to Maple, which produces the (correct) solution:

```
                2  2
               X  Y  + X Y + 1
      --------------------------------
              2  3        2  2
       (1 - X  Y ) (1 - X  Y ) (1 - X)
```

3.1.3 Comments

It is clear that this is not the best way to approach the output or, indeed, the general method. Firstly, it is not efficient to pass ever-increasing strings about throughout the recursion process. Also, the output is very Maple-oriented, which is an undesirable bias.

This is improved substantially later on. It was chosen at this stage as a way of ensuring that the integration process worked successfully. However, before the output process was modified, the input process was written.

3.2 Parser

Having a program that solves integrals is obviously a good start, but it would be nice to able to give it integrals in the first place, rather than a cryptic string of numbers. The fairly rigid format of the integrals allowed for a straightforward recursive descent parser.

It would also be convenient to be able to surround integrals in the input with other text, such as Maple commands. For this to be possible, there has to be a way of delimiting integrals, and it was decided that they should begin with "`Int`" and end with a full stop.

The syntax chosen for an integral is as follows. Note that the character '`|`' does mean that character, representing either 'divides' or a valuation; different choices are on their own lines. 'ϵ' represents the empty string.

Integral ⟶ `Int` *Integrand* `dv` *Conditions* `.`

Integrand ⟶ ϵ
`|` *Variable* `|` *Integrand*
`|` *Variable* `|^`*Exponent Integrand*

Exponent ⟶ valid sum of integer multiples of s and integers[8]
{ *Exponent* }
`(` *Exponent* `)`

Conditions ⟶ ϵ
Monomial `|` *Monomial*
Monomial `|` *Monomial* `,` *Conditions*

Monomial ⟶ ϵ or 1
Character Monomial
Character `^`*Integer Monomial*
{ *Variable* } *Monomial*
`(` *Variable* `)` *Monomial*
{ *Variable* }`^`*Integer Monomial*
`(` *Variable* `)` `^`*Integer Monomial*

The integers which occur as exponents in the conditions section are assumed to be positive. This is acceptable since a negative exponent on the left hand side of a condition should instead have been written as a positive exponent on the right.

[8]For example, "$s - 1 + 3s + 4 - 2s$". The expansion of the syntax of this has been omitted, as the form is straightforward. No parentheses or unary minus (except at the beginning) are permitted.

There is no expansion for *Variable* above. This is because these are not interpreted directly by the parser. For the integrand, everything occurring between two '`|`' characters (taken in pairs) is called a variable. For the conditions, the delimiters are a pair of brackets. Text thus contained is then checked to see if it is valid. For an expression to be a valid variable, it must begin with an alphabetic character and contain only alphanumerics and the underscore and prime ($'$) characters.

Enclosing every variable between delimiting characters would be excessive, although it is necessarily the case for the integrand because of the valuation bars. For the conditions, a single alphabetic character is read directly as a variable. Variables with more than one character to their name must be enclosed within brackets.[9]

A monomial in a condition may also be left empty, in which case it will be taken to be the unit 1. An entirely empty condition is also permitted, since this will allow for the input of integrals with no conditions.

Inspection of the description of an exponent above shows that the brackets are optional. This is intentional and for simplicity when writing, say, `|x|^s`. This does allow for expressions such as `|x|^s-1|y|`, which will be treated as `|x|^{s-1}|y|`. This may seem incorrect for the usual orders of precedence, but is allowed here. Since everything within an integral is multiplicative, there is no other plausible interpretation than the above. The user may of course use brackets if preferred.

Should a variable, exponent or monomial fail to conform to the correct format, the user is told which is invalid and why. For example, the messages "Bad Variable: ⟨*string*⟩" or "Exponent Error: ⟨*string*⟩" tell the user where the error is. No attempt is made by the program to find a sensible interpretation for incorrect input, since there may be a number of possibilities.

A comments facility is included as well, since a user may wish to note where certain parts of an integral come from. A comment is enclosed between `/*` and `*/`, and everything between these is ignored when read, so

[9] This inclusion of variable names in the input necessitates an extra piece of information being stored in the Integral class. Before, when displaying integrals, the variables were assigned as $a, b, \ldots$, but now a record is kept of the user's choice of names.

nested comments should be avoided. Should a comment be left unterminated, the user is warned since it is likely to be unintentional, for it means that everything from the opening `/*` to the end of the file will be ignored.

With this parser in place, input can now be read. This may be from a file or taken from standard input, using a BufferedReader. Supposing that the input contains N integrals with surrounding text, the text is stored in a size $N+1$ String array, and the integrals are parsed and stored in a size N Integral array.

The N integrals are then solved in turn, with the Maple string output from each being placed appropriately amongst the $N+1$ stored strings. This results in an output file which is same as the input file, but with the integrals within it replaced by their evaluation. This is effectively a preprocessing step before applying Maple to sum the integrals and apply any instructions included in the input text.

Example. Consider feeding the program the three parts of the Heisenberg integral separately, via the file:

```
assign(a, Int |x|^{s-1}|y|^{s-2}|z|^{2s-3} dv
          z|xzy.
      );
assign(b, Int |x|^{2s-3}|y|^{s-2}|z|^{s-3} dv
          xz|xy.
      );
assign(c, Int |x|^{2s-3}|y|^{s-2}|z|^{s-3} dv
          xz|xy,
          z|1.  /* this means z is a unit */
      );
simplify(a+b-c);
```

As described above, the these integrals are parsed and intepreted, then solved. The strings produced as the solution are substituted, thus:

```
assign(a,1/((1-X)*(1-X*Y)*(1-X^2*Y^2)));
assign(b,1/((1-X^2*Y^2)*(1-X*Y)*(1-X^2*Y^3)));
assign(c,1/((1-X^2*Y^2)*(1-X*Y)));
simplify(a+b-c);
```

3.3 Simplifier

So far, the program will take an integral or series of integrals as input, solve them, then pass the results to Maple to sum together and simplify. As will be covered in more detail in the Evaluation chapter, the solving part is fairly quick, but the simplification by Maple takes a lot longer. This is perhaps not surprising, for it is being passed a large, nested string and then asked to perform a lot of calculation.

An improvement would therefore be to do some of this work beforehand. Maple is missing some insight from the theory – it will be using the full force of a general algorithm for adding rational functions and factorising general polynomials, but these functions are of a very precise form. For example, it should be easier to test for factors of the form $(1 - X^aY^b)$ than to apply a more general factoring algorithm. Having this sort of simplification being done by the main program would also mean less dependence on Maple, which is another desirable goal.

The Simplifier module was written to accept the output of the Solver, which means that it originally accepted the string that was to be passed to Maple. Once the Simplifier was working successfully, the inelegant string set-up was removed, but that will be covered in the next section. For now the string input shall be assumed, but that only matters for the initial step.

3.3.1 Data Structures

In the Solver, there was a description of how to encode a monomial containing some unfixed number of variables, and storing each as an array of integers was deemed sufficient. Here, there must be a way of encoding polynomials in the two variables X and Y. Using a similar method could end up with a large two dimensional `int[][]` array, where the (m, n)-th entry is the coefficient of X^mY^n. This is undesirable for a number of reasons:

- negative indices would not be possible. This would cause problems with any X^mY^{-n} terms, which could arise from an integral such as
  $$\int |x|^s d\nu = (1 - p^{-1})/(1 - p^{-s-1}) \implies 1/(1 - XY^{-1}).$$

- there are likely to be very sparse matrices. For example, $1-X^{100}Y^{100}$ is a matrix with over ten thousand entires, only two of which are non-zero.
- there would be a fixed size for each column, which is likely to be wasteful, given that the Y-exponents could range differently for each X-exponent.
- keeping lists of polynomials could begin to involve three-dimensional arrays, imposing unwanted length constraints.

A better data structure was clearly needed to store polynomials, and the following was chosen. Define the class *Singleton* as a way of storing a term aX^mY^n, keeping the values a, m, n in three separate data fields. Define the class *Polyton* (a curious word which has stuck) whose primary data field is an array of Singletons.

This enables a polynomial to be stored without wasted space as a Polyton, and a collection of polynomials as a Polyton array. Internally, this does have a three-dimensional form, but with varying sizes of entries permitted.

Before being able to manipulate them, the strings from the Solver module must be encoded as Polytons. This is done by a scan through the algebraic expressions in the string converting single terms into Singletons, and then turning bracketted sets of Singletons into Polytons. Detail shall be omitted here, since the string approach is removed later on. For now, it will be assumed that there is a set of rational functions to be summed and multiplied and so on.

3.3.2 Algorithms

The output of the Solver gives the polynomials in a lexicographic order, in which X^kY^l occurs before X^mY^n if and only if either $k < m$, or $k = m$ and $l < n$. This is not surprising, since the output consists of terms of the form $(1 - X^mY^n)$, and m is never negative.[10] Constructing algebraic operations

[10] The only anomaly is the term $(1 - Y^{-1})$, but this is made ordered by a simple swap.

which preserve order will remove the need for any sorting afterwards, and enable swifter analysis.

The first operation to construct on Polytons is addition. Using the two-dimensional array format rejected earlier, addition would consist of scanning over the matrix and adding entries – an $O(a^2 + b^2)$ approach, where a, b are the largest exponents in the two summands. However, addition of Polytons may be performed in $O(c+d)$ time, where c, d are the lengths of the Singleton arrays stored in the Polytons. In the worst case, when the Polyton has every pair of exponents, these are the same, but in general the latter method will be much faster.

Since the two Polytons to be added are assumed to be in sorted internal order, they may be added swiftly in a manner familiar from, say, Mergesort. Compare the smallest element of each, and move the smaller to the result. (If the two are equal in exponents, add their coefficients together in the result.) This is continued with the remaining terms in the summands, until both are exhausted.

This is clearly order-preserving, and yields a Polyton of size at most the sum of the sizes of the two original Polytons. Subtraction can be included easily, by having a flag indicating whether to add or subtract the coefficients.

Multiplication may be performed using the two-dimensional array approach in time $O(a^2b^2)$. Using Polytons and the addition function above, the task may be completed in $O(cd)$, essentially performing long multiplication. Multiplying a Polyton by an individual Singleton is easy – it requires a single pass through the Polyton, incrementing the exponents by those in the Singleton and multiplying the coefficients.

To multiply two Polytons, proceed through one a Singleton at a time, at each step multiplying it by the other and adding it to a running total. Each pass is $O(d)$, and each addition $O(2d)$, and this happens c times, giving an algorithm in time $O(cd)$, which is also order-preserving.

3.3.3 Application

Given the output of an Integral in the form of expressions nested in a string, the Simplifier would first find an innermost expression, that being one enclosed in a pair of brackets with no other brackets between – the result of a bottom level of recursion. This will be a sum of three terms, coming from the three parts of a blow-up. The Simplifier adds these terms together using the above operations, then substitutes the single term as a string in the original expression. This is then repeated, until there are no more 'inner' expressions, at which one point the string is one large rational function.

The algebra required at each stage is the same: adding expressions of the form

$$\frac{N_1}{D_1} + \frac{N_2}{D_2} - \frac{N_3}{D_3},$$

where the N_i and D_i are polynomials. The N_i are initially 1, and the D_i are products of the form $(1 - X^{m_1}Y^{n_1})\dots(1 - X^{m_k}Y^{n_k})$, as seen before. Accordingly, such terms are stored as a pair of a Polyton and a Polyton array. It is desirable to preserve the factored format of the denominator, and so these remain stored as an array, rather than being expanded into a single Polyton.

The summation takes the terms in pairs, from the left. The first optimisation is to extract any common factors shared by D_1 and D_2, so that there are fewer terms left to be cross-multiplied with the N_i. This is one reason for keeping the denominators in their factored form, and is done by picking the entries of D_1's Polyton array in turn and comparing each with those in D_2. Any shared Polytons are put into a third Polyton array, D, and are removed from both D_1 and D_2. We then have

$$\frac{N_1}{D_1} + \frac{N_2}{D_2} \;=\; \frac{1}{D}\left(\frac{N_1}{D_1'} + \frac{N_2}{D_2'}\right) \;=\; \frac{1}{D}\left(\frac{N_1 D_2' + N_2 D_1'}{D_1' D_2'}\right)$$

The numerator here is calculated by two applications of the Polyton *multiply* method, followed by one application of Polyton *add*. The denominator

of the sum is formed by concatenating the three Polyton arrays together.

This method is then repeated with this summed expression and the third term above, with a subtraction being performed this time.

Once done, the result is a single expression, in the form of a pair of a Polyton and a Polyton array, as were the original terms. These may then be converted back to a string, to be substituted in the input string. After repeated application, the result is a string representing a single pair of a Polyton and a Polyton array. This is a valid string to pass to Maple.

Example. The Heisenberg integral again. The terms to add together have the three N_i being the Polyton for 1, and the D_i Polyton arrays encoding

$$\begin{aligned} D_1 &= \{(1-X),(1-XY),(1-X^2Y^2)\}, \\ D_2 &= \{(1-X^2Y^2),(1-XY),(1-X^2Y^3)\}, \\ D_3 &= \{(1-X^2Y^2),(1-XY)\}. \end{aligned}$$

The check for shared denominators between D_1 and D_2 gives

$$D = \{(1-XY),(1-X^2Y^2)\}.$$

The numerator of the sum is the Polyton

$$\begin{aligned} N_{12} &= N_1D_2' + N_2D_1' \\ &= (1)(1-X^2Y^3) + (1)(1-X) \\ &= (2-X-X^2Y^3), \end{aligned}$$

and the denominator is the concatenation of the arrays D, D_1, D_2, as

$$D_{12} = \{(1-XY),(1-X^2Y^2),(1-X),(1-X^2Y^3)\}.$$

The check for shared denominators between D_{12} and D_3 gives

$$D' = \{(1-XY),(1-X^2Y^2)\},$$

once again. The numerator of the sum is the Polyton

$$\begin{aligned} N_{123} &= N_{12}D_3' + N_3D_{12}' \\ &= (2-X-X^2Y^3)(1) - (1)\left((1-X)(1-X^2Y^3)\right) \\ &= (1-X^3Y^3), \end{aligned}$$

and the denominator is the concatenation of the arrays D', D_{12}, D_3 as

$$D_{123} = \{(1 - XY), (1 - X^2Y^2), (1 - X), (1 - X^2Y^3)\}.$$

This gives the result

$$\frac{(1 - X^3Y^3)}{(1 - X)(1 - XY)(1 - X^2Y^2)(1 - X^2Y^3)}.$$

3.3.4 Factorisation

The summation produces a string that can be passed to Maple. Although it now a summed expression, it is likely that some factorisation will be required. It is common to find that the smaller parts formed by a blow-up have terms in their denominators which cancel off when the parts are all summed together. However, as mentioned earlier, knowing the form of the potential factors gives an advantage over Maple – it tries a general purpose factorising algorithm, but factors may often be found by testing each of those in the denominator to see if any divide the numerator.

At this point, consider a single expression of the form

$$\frac{N}{D_1 \ldots D_k},$$

where N is a general Polyton and the D_i are stored in a Polyton array, with each being of the form $(1 - X^{m_i}Y^{n_i})$. To cancel off a factor from the denominator, it should first be shown to divide the numerator, since testing for division is a far cheaper operation than dividing directly.

If $(1 - X^mY^n)$ is a factor of N, then replacing every power of X^mY^n in N by 1 should leave a Polyton which sums to 0. This replacement may be done in a single pass over N. If the sum afterwards is not zero, $(1 - X^mY^n)$ is not a factor and this term may be left in the denominator array. The testing then proceeds with the next denominator term.

If a factor is found through getting a zero sum, it should be divided into the numerator. This is done by long division, subtracting off appropriate multiples of the factor until the dividend is zero, building the quotient gradually. This quotient replaces the numerator, and the factor is removed from the denominator array.

This is repeated for all of the factors in the denominator, after which the numerator will usually be much smaller. At this point, it seemed acceptable not to attempt to find further factors in the numerator, since this would require general factorising with which Maple is better equipped to cope. Also, experience at performing these integrals has shown that further factorisation rarely happens, and the output of this factorisation algorithm is likely to be the final answer.

One optimisation that may be applied here is to test the denominator factors in decreasing order, where the ordering on terms $(1 - X^m Y^n)$ is determined by the lexicographic order on the Singletons $X^m Y^n$. There are two reasons for this:

- dividing the numerator by a larger factor is likely to give a greater reduction in its size, making later passes through it faster.
- suppose that $(1 - X^2)$ is a factor, but $(1 - X)(1 - X^2)$ is not. If the $(1-X)$ factor is cancelled off first, then when trying the $(1-X^2)$ term, it will not divide and so will be left. This will result in the numerator factor $(1 + X)$ not being cancelled. Testing the denominator terms in decreasing order will avoid problems of this sort.

Example. Consider the result for the Heisenberg integral:

$$\frac{(1 - X^3Y^3)}{(1 - X)(1 - XY)(1 - X^2Y^2)(1 - X^2Y^3)}.$$

Testing begins with the denominator term $(1-X^2Y^3)$, and so every power of X^2Y^3 in the numerator is reduced to 1. This means the numerator becomes $(1 - X)$, which is clearly not 0, so this is not a factor. Similarly, $(1 - X^2Y^2)$ leaves the reduced numerator $(1 - XY)$, so this is not a factor either.

However, when trying $(1 - XY)$, the numerator reduces to 0, and so this is a factor. It is divided off, leaving the numerator $(1 + XY + X^2Y^2)$. Testing the final term, $(1 - X)$, does not give zero. So, the factorised solution is:

$$\frac{(1 + XY + X^2Y^2)}{(1 - X)(1 - X^2Y^2)(1 - X^2Y^3)}.$$

3.4 Trees

As has been mentioned before, having the interface between the Solver and Simplifier modules use strings is not very efficient. It served when output from the Solver was passed directly to Maple, but has now been replaced. This section describes the re-writing of parts of the Solver and Simplifier into a much more sensible arrangement.

By the nature of blow-ups, the solving process creates a tree. The main integral has three child integrals, and so on recursing downwards, therefore each integral may be associated with a node in a ternary tree. The information such a node contains is as follows:

- references to its three children
- a local integral – that to be solved in this subtree
- a local record of the solution of the subtree below

During the calculation, some node will contain an integral that needs solving. The child nodes will not have been created yet, so the references will be null and the solution equal to 0. Performing a blow-up creates three child nodes, which each get passed a variant on the current stored integral. Once this has been passed on, the local integral may be assigned the value null, to free up space.

Eventually, the process reaches a node which does not need blowing up, containing a condition-free integral that can be solved directly. This solution

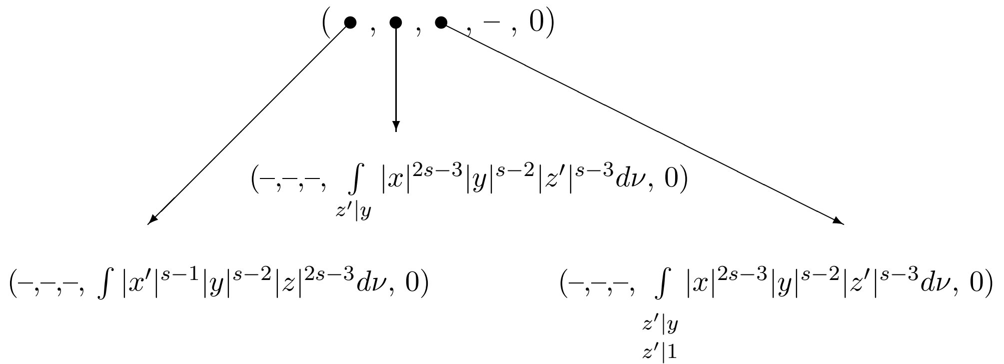
is placed into the local record in the form of a Polyton for the numerator and a Polyton array for the denominator. The local integral may again be set to null.

Once all the integrating has finished, the tree is complete. The non-leaf nodes contain just references to other nodes; the leaf nodes contain algebraic expressions, stored via Polytons.

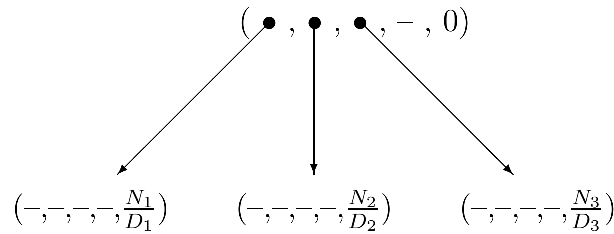
Example. The Heisenberg integral again. Initially, there is just the root node (using ‘ – ’ to represent *null*) :

$$(- , - , - , \int_{z|xy} |x|^{s-1}|y|^{s-2}|z|^{s-3}d\nu, 0)$$

A blow-up with x and z yields the following:



The left hand branch may be solved directly, as before. The others undergo more variable changes (such as $y \to y'z$) but are not blown up. They are then solved in place, yielding the tree:



where the N_i at this point are Polytons representing 1, and the D_i are Polyton arrays encoding the denominators seen earlier.

This avoids having to pass large strings around during calculation, and also leaves the data in a much easier format for simplifying. The integration process is largely unaltered – the method which turned the Monomial arrays into strings was replaced by one which created the denominator Polyton arrays, and the recursive solving method created three new nodes at each stage rather recursively returning the solution to the three sub-integrals.

With a tree in the solved format, it may be printed out directly, as for the original Solver. This may be done by a simple recursive print method, along the lines of:

node.print() : *if subnode1 is null, return this node's stored solution*
else return "`simplify(`"+*subnode1.print()*+"`+`"+...+"`)`";

Only one subnode need be tested for being null, since an integral was either blown up into three parts, or left alone.

It is also easy to apply the Simplifier to this tree structure. It no longer has to search through a string to find an "innermost expression", since these will be formed by the last-but-one level of the tree, where three terms are added together. Accordingly, the tree may be summed in a manner similar to the printing method above.

When the summation is finished, the result is a single node once again. Its three pointers and stored integral are null, and the local solution is the sum of the entire tree, still stored in Polyton format. This may either be output directly as a string for Maple, or be factorised first. The factorisation algorithm is as before, as it worked on Polytons, not strings.

In addition to being a far faster and neater approach, using the tree structure has the advantage that it is less Maple-oriented than the string version. Here, bias towards Maple will only occur after all the integrating, when printing the tree. One could easily have other short print methods bound to other algebra packages.

Chapter 4

Evaluation

Each part of the code was tested at the time it was written. The testing for correct complete integration used a set of known p-adic integrals taken from a variety of sources, such as [4], [9] and [10]. For the individual modules, the later ones assume that the previous output is correct, although they also contain checks in case something has gone wrong.

4.1 Parser

The Parser was tested by being fed a collection of integrals either conforming to or failing to conform to the syntax described in Section 3.2. Those that were valid integrals were interpreted correctly, and the following example demonstrates most aspects of the syntax.

Example. Given the input file

```
Int |x_1|^(s-1)|y|^{2s-2+3-s-2}|z|^2s-1 dv
z|{x_1}^2y,   /* a comment about giraffes */
a^13|y(z).
```

the Parser correctly produces

```
                 s-1   s-1   2s-1
     Int  |x_1|   |y|   |z|     dv

   z | (x_1)^2 y
a^13 | y z
```

Variants on the above were tested as well. For example, splitting the integrand, conditions or comments across lines, or placing comments in the middle of terms. These were all parsed successfully.

Input can fail to conform to the syntax for various reasons, such as the following.

Examples. (i) The input “`Int |x*y|^s dv.`” provokes the response “`Bad Variable: x*y`”.

(ii) Input “`Int |x_1|^{s-1} dv x_1|y.`” causes the response “`Bad Variable: _`”. (Recall that names of more than one character in a condition must be enclosed in brackets.)

(iii) Input “`Int |x|^{s dv.`” provokes an “`Exponent Error`” message.

(iv) The input “`Int |x|^s dv /* stuff`” causes an “`Unclosed Comment`” message.

In addition to individual integrals, files containing a number of integrals amidst surrounding text were tested. These were all parsed correctly (or produced the correct errors if appropriate), but no examples are included here, as they would look much like those above.

The output of the Parser stage is a series of Monomial and Integral objects which are then passed to the Solver, and the displayed output above indicates that these are correct. Therefore, it shall be assumed that the input to the Solver section is correct. (By the nature of how the Solver is called, it will always be a *valid* integral – the assumption is that it is the *correct* one.)

4.2 Solver

The requirement for the Solver module is to take an integral and produce solved output. This was more directly testable for the original version which used strings throughout rather than creating a tree, since its output could be examined more immediately. With the tree version, tests were required throughout to ensure that the tree-making process was working, as well as the output stage.

4.2.1 String version

The first stages were checked by displaying the steps taken at each blow-up – that is, which variables it used, and what the sub-integrals were. This would enable checking that each was performed correctly, and that it had progressed on to the appropriate sub-integral. Once blow-ups were seen to be performed successfully, this extra displayed information was removed, since for larger integrals it can cause many screenfuls of extra text.

The conversion from internally stored information to strings suitable for Maple was tested by feeding it condition-free integrals with a variety of exponents.

Example. The input "`Int |a|^{s}|b|^2|c|^{2s+1}|d|^{3s-2} dv .`" produces the output string:

```
simplify(1/((1-Y^(-3))*(1-X*Y^(-1))*(1-X^2*Y^(-2))*(1-X^3*Y))).
```

Note that where the input exponents are $as - b$, those in the output are $as + b - 1$, consistent with the derivations earlier.[1] Also note the appropriate handling of exponents equal to 0 or 1.

For more general integrals, see Section 4.4.

[1]Recall that $X = p^{-s}, Y = p$.

4.2.2 Tree version

The integration steps and blow-ups were unchanged between the two versions. The testing here was concerned with the tree structure. This was checked for correctness by generating the Maple string by a simple recursive printing method and comparing to that obtained for the equivalent integral by the string version of the program.

The tree structure also allowed for closer examination. A particular subtree could be look at in detail, rather than just considering the whole integral in one go.

For more general integrals, see Section 4.4.

4.3 Simplifier

The Simplifier was first tested by asking it to add or multiply sets of polynomials encoded as Polytons. These test polynomials were chosen to cover a variety of possibilities, such as negative indices, terms cancelling, multiple terms with the same exponents, and so on. The order preserving nature meant that the results could be checked easily.

For the factorisation algorithm, each part was tested in turn. Testing for factors of the form $(1 - X^m Y^n)$ involved using the multiplier to create large examples with various factors of this form, then performing the test for division described earlier. Division was tested similarly, with the results being compared to the original terms.

Essentially, this process consisted of evaluating the expression

$$\texttt{subtract}\left(\mathit{Polyton}\ N,\ \frac{\mathit{Polyton}\ N \times D_1 \ldots D_i}{\mathit{Array}\ (D_1, \ldots, D_i)}\right)$$

and checking that the result was zero, for a variety of Polytons $N, D_1, \ldots, D_i$. This was performed by a small automated procedure, being fed a number of test expressions.

4.4 General integrals

To test that the program was actually solving integrals correctly, first a number of known results were used, and then more complicated examples, whose results were previously unknown. These cannot be checked for exact accuracy without doing the calculation by hand, but there are theoretical properties[2] that an answer should have that can be examined, and this was done. Also, integrals were tested on different versions of the program to ensure that the same solution was produced – for example, using the original string version or the tree version, and with or without summation and factorisation.

Example. The Heisenberg integral. Given the input file

```
Int |x|^{s-1}|y|^{s-2}|z|^{s-3} dv
z|xy.
```

the factorised solution produced is the correct

```
(1+X*Y+X^2*Y^2)/((1-X)*(1-X^2*Y^2)*(1-X^2*Y^3)).
```

Example. The above example is too short to illustrate the benefits of summation and factorisation. The following example occurred during the calculation of the zeta function for a particular Lie algebra in [9]. It is the difference of two integrals. The input file is:

```
a := Int |u|^{s-1}|w|^{2s-4}|x|^{2s-3}|y|^{s-4}|z|^{2s-5} dv
      y|uwx^2z, y|ucz,
      y|uwx^2e, y|uce. ;

b := Int |u|^{s-1}|w|^{2s-4}|x|^{2s-3}|y|^{s-4}|z|^{2s-5} dv
      u|1,
      y|uwx^2z, y|ucz,
      y|uwx^2e, y|uce. ;

i:=simplify(a-b);
```

[2]For example, all known solutions to integrals corresponding to Lie algebras satisfy a functional equation of a particular form.

This may be used to compare the size of the output of various applications of the program. Using the original version without summation, the output file is a nested string of size approximately 18K. However, using summation, but not factorisation, produces the un-nested output:

```
a:=(1-X^3*Y^5-X^4*Y^4-X^4*Y^5-2*X^4*Y^6-X^4*Y^7-X^5*Y^6-2*X^5*Y
```

... *twenty-five similar lines* ...

```
(1-X^4*Y^6)*(1-X^4*Y^6)*(1-X^4*Y^7)*(1-X^5*Y^9)*(1-X^6*Y^10)));

b:=(1-X^3*Y^5-2*X^5*Y^7-X^5*Y^8-X^5*Y^9-X^6*Y^8-2*X^6*Y^9-X^6*Y
```

... *nine similar lines* ...

```
(1-X^3*Y^5)*(1-X^3*Y^6)*(1-X^4*Y^6)*(1-X^5*Y^9)*(1-X^6*Y^10)));

i:=simplify(a-b);
```

This is a file of size about 3K, and was derived using internally selected blow-ups. Running the same again, this time entering a particular set of blow-ups, reduces this to about 1K.

Finally, the factorisation algorithms may be applied, giving the 300b answer:

```
a:=(1+2*X^2*Y^3+X^3*Y^4-X^4*Y^4-X^4*Y^5+2*X^4*Y^6-X^5*Y^6-X^5*Y
^8-X^6*Y^8-X^7*Y^11)/((1-X)*(1-X^2*Y^2)*(1-X^2*Y^4)*(1-X^3*Y^5)
*(1-X^3*Y^6)*(1-X^4*Y^6)));

b:=(1+X^2*Y^3+X^3*Y^4+X^4*Y^6-X^5*Y^7-X^5*Y^8-X^6*Y^9-X^7*Y^11)
/((1-X^2*Y^2)*(1-X^2*Y^4)*(1-X^3*Y^5)*(1-X^3*Y^6)*(1-X^4*Y^6)));

i:=simplify(a-b);
```

The factorised answer is the same regardless of the choice of blow-ups used to solve the integral, which is what is to be hoped for. This is a good check that the various methods are consistent. Any of these files may be fed to Maple, giving the final (correct) solution:

$$\frac{X(1 + XY^3 + X^2Y^3 - X^3Y^5 + X^3Y^6 + X^4Y^7 - X^5Y^7 - 2X^5Y^8 + X^5Y^9 - X^6Y^9 - X^7Y^{11})}{(1 - X)(1 - X^2Y^2)(1 - X^4Y^2)(1 - X^3Y^5)(1 - X^3Y^6)(1 - X^4Y^6)}$$

This example suggests an improvement that could be made to the current state of the project. One of the purposes of the introduction of summation and factorisation was to remove dependence on algebra packages, yet the example above resorts to more Maple. It features the addition and simplification of two integrals, which is precisely the sort of manipulation these features can handle. However, there is no facility for applying them *outside* of an individual integral calculation. Allowing addition and multiplication of this nature would not be difficult and would mean much greater functionality, but unfortunately this wasn't thought of until it was too late.

4.5 Efficiency

There have been various choices of methods used to solve integrals – strings or trees, factorising in the program or leaving it to Maple – but which is most efficient? Making the process more independent by introducing the Simplifier is good and helpful, but it should be checked that it does not make things significantly slower. Fortunately, it does not.

The examples earlier in this chapter are sufficiently small that the differences in each version are negligible, and so the following integral is used as an efficiency test. It is the "unpleasant" example mentioned in Chapter 2, which also appeared as part of a Lie algebra calculation in [9].

$$\int_{\substack{y \,|\, ew \\ y \,|\, uwx^2 \\ z \,|\, xy \\ y \,|\, uxa \\ z \,|\, x^2c \\ y \,|\, exa}} |u|^{2s-5}|w|^{2s-4}|x|^{2s-2}|y|^{s-4}|z|^{s-4}d\nu$$

The times for computation are approximate, based on a number of runs. The machine used for each of these was one of the CUS machines mentioned earlier – a Sun Enterprise 420R using a 450 MHz UltraSPARC-II CPU, running version 1.2.1 of Java.

Method of calculation	Solution time	Maple time
By hand	∼2 weeks	5m 40s
String version, no simplifying	3s	6m 40s
Tree version, no simplifying	1s	6m 40s
String, simplifying, not factorising, unspecified blow-ups	1m50s	16s
Tree, simplifying, not factorising, unspecified blow-ups	48s	16s
Tree, simplifying, not factorising, user-specified blow-ups	8s	7s
Tree, simplifying, factorising, user-specified blow-ups	16s	—

The majority of the results here are not surprising. Direct solving of the integral is fast, but the output is large and takes Maple a relatively large amount of time add together. Simplifying and factorising slow the program down, but the output is many times smaller and may be processed much faster.

There are two points to note:

- Without the simplifier, the total time is a little under seven minutes (1-3s to solve, then 6m40s for Maple), but including it reduces this to approximately one minute (48s to solve, then 16s for Maple). This is through taking advantage of the forms of the terms involved.
- The time taken when employing the factorisation algorithm is much the same, in total, as that for when Maple was used to factorise. This is not the speed increase that was hoped for. However, it is not slower and is independent of Maple, which is a good thing.

Chapter 5

Conclusions

As discussed in Chapter 2, the original project plan was changed somewhat to give the project a more computational nature. The revised plan, while not written down as formally as that submitted, was followed fairly closely. Work on the project deviated from the revised plan in the weeks before the exams, as more time was devoted to revision.

With hindsight, there are some parts of the project that could have been done differently. Since it was decided to use Java in preference to ML, the time spent reading up on ML could have been put to other uses. Also, time could have been better spent using trees rather than strings from the start. However, both of these are said with the hindsight and knowledge of a year of lectures, and so are not a criticism of the original approach and work.

5.1 Improvements

There are various ways in which the program could be extended and improved. Below is a list of these – things that would most likely have been included if they had occurred earlier in the year.

- It was noted in Section 4.4 that it would be useful to be able to manip-

ulate *solved* integrals using the Polyton algebra of the Simplifier. This would mean even less dependence on an algebra package for computing final results. This would not be difficult to do, since the addition and multiplication methods already exist – it would require modifications to the Parser to detect such commands, and to the Simplifier to apply them.

- Currently, when not given by the user, the variables used in blow-ups are chosen to be optimal for a given condition, in order to remove it as swiftly as possible. As the increase in speed judicious choices can bring demonstrates, this is not necessarily the best choice across all the conditions. Since the time taken to solve the integrals is small compared to that to sum the result, it would be possible to try different pairs of variables to find a better selection as the automated pair.

- Typing this dissertation in LaTeX [5] has suggested that a useful feature would be to have the choice of output in LaTeX format. It would save much fiddling! The input could still be in the same text format, with that also being converted to LaTeX as well as the result. Similarly, a choice of output in MathML [8] format would be nice.

- It would be nice to have a more elegant user interface. The program is run via `java Project filename`, or via `java Project -f filename` to apply factorisation, but a possibility would be to have a nice menu with options to factorise and so on.

5.2 Uses and possibilities

The project has already proved useful – one of the earliest working versions of the Solver managed to find a typing mistake in my Ph.D. thesis a few weeks before my *viva voce* examination. This enabled a correction to be made in time and also impressed the examiners!

One obvious extension to the available features would be some facility to cover polynomial integrals. The structures of Singletons and Polytons could be used to represent conditions in such integrals, since the Monomial class would no longer be sufficient. This is unlikely to be that successful for some

time, unfortunately, until more methodical theory is established. For, while polynomials are reducible, their reduction is sometimes of a rather *ad hoc* nature.

In the mean time, however, being able to compute monomial p-adic integrals will speed up the theoretical side of things, and so the project will assist in writing its own extension.

Bibliography

[1] G. Bodnar and J. Schicho. *A computer program for the resolution of singularities*, in H. Hauser, editor, *Resolution of Singularities*, Birkhäser, 2000

[2] J. Denef, *The rationality of the Poincaré series associated to the p-adic points on a variety*, Invent. Math. **77** (1984), 1-23

[3] D. Grenham, *Some topics in nilpotent groups*, D.Phil. thesis, University of Oxford, 1988

[4] F.J. Grunewald, D. Segal, G.C. Smith, *Subgroups of finite index in nilpotent groups*, Invent. Math. **93** (1988), 185-223

[5] L. Lamport, *LaTeX: a Document Preparation System*, Addison-Wesley, 1994

[6] *The Magma Computational Algebra System*, retrieved July 2002: http://magma.maths.usyd.edu.au/magma/

[7] *Waterloo Maple* , retrieved July 2002: http://www.maplesoft.com/

[8] *W3C MathML* , retrieved July 2002: http:/www.w3.org/Math/

[9] G.L. Taylor, *Zeta Functions of Algebras & Resolution of Singularities*, Ph.D. thesis, University of Cambridge, 2001

[10] J. White, *Zeta functions of groups*, D.Phil. thesis, University of Oxford, 2000

[11] S. Wolfram, *The Mathematica Book*, Cambridge University Press, 1999

Appendix A

Lie Algebras and Zeta Functions

While the theory which inspired this project is rather complicated, the majority of it is not required. Below is an informal description of where these integrals come from and why they are worth solving.

Let G be a finitely generated group, and define $a_n(G)$ to be the number of subgroups of G of index n. Define the *zeta function* of G to be

$$\zeta_G(s) \;=\; \sum_{H \leq G} |G : H|^{-s} \;=\; \sum_{n=1}^{\infty} a_n(G) n^{-s}.$$

This function was introduced in a paper [4] by Grunewald, Segal and Smith in 1988, as a way of encoding subgroup growth in an analytic form. The same paper showed that if G is nilpotent, then there is an Euler product decomposition

$$\zeta_G(s) = \prod_{p \text{ prime}} \zeta_{G,p}(s),$$

where $\zeta_{G,p}$ is the *local zeta function*, defined to be

$$\zeta_{G,p}(s) = \sum_{r=0}^{\infty} a_{p^r}(G) p^{-rs},$$

which is a sum over subgroups of p-power index. If G is also torsion-free, then these local zeta functions at each prime are rational functions. That is,

$$\zeta_{G,p}(s) = \frac{\Phi(p, p^{-s})}{\Psi(p, p^{-s})},$$

with $\Phi(x, y)$ and $\Psi(x, y)$ polynomials over $\mathbb{Z}$. In particular, it follows from a theorem of Denef [2] that the denominators of the local zeta functions may be expressed in terms of linear factors, as

$$\Psi(x, y) = (1 - x^{b_1} y^{a_1}) \ldots (1 - x^{b_r} y^{a_r}),$$

for some r, where the $a_i, b_i \in \mathbb{N}$.

Grunewald, Segal and Smith also showed that such local zeta functions may be expressed as p-adic integrals by considering the possible bases a subgroup of G can have.

Let $\{x_1, \ldots, x_d\}$ be a basis for G. Then, a subgroup H of G is generated by $\{X_1, \ldots, X_d\}$, where

$$X_i = \sum_{j=i}^{d} m_{ij} x_j$$

for some $d \times d$ matrix $M = (m_{ij}) \in T_d(\mathbb{Z}_p)$, where $T_d(\mathbb{Z}_p)$ is the set of $d \times d$ upper-triangular matrices over $\mathbb{Z}_p$, the p-adic integers.

It is shown in [4] that

$$\zeta_{G,p}(s) = (1-p^{-1})^{-d} \int_{M \in \mathcal{M}_p} |m_{11}|_p^{s-1} |m_{22}|_p^{s-2} \dots |m_{dd}|_p^{s-d} d\nu_p,$$

where $\mathcal{M}_p$ is the set of matrices which gives a basis as described above, such that H is indeed a subgroup. Here, $|.|_p$ denotes the p-adic absolute value, and ν_p is the additive Haar measure on $T_d(\mathbb{Z}_p)$.

It may be shown, by considering the requirements on the basis elements X_i for H to be a subgroup, that the set $\mathcal{M}_p$ may be described by a set of conditions of the form $v\left(f(\mathbf{x})\right) \leq v\left(g(\mathbf{x})\right)$, where f is a product of diagonal elements of the matrix, and g is a homogeneous polynomial in any of the elements.

These integrals are *cone integrals*.

Appendix B

The Heisenberg Lie Algebra

Throughout the dissertation, the integral for the Heisenberg Lie Algebra is used as an example. Below is a derivation of the integral from the definition of the algebra, for those interested.

The Heisenberg Lie Algebra is defined to be that consisting of 3×3 strictly-upper-triangular matrices, thus:

$$H = \begin{pmatrix} 0 & \mathbb{Z}_p & \mathbb{Z}_p \\ & 0 & \mathbb{Z}_p \\ & & 0 \end{pmatrix}$$

This has basis elements

$$\begin{pmatrix} & x & \\ & & \\ & & \end{pmatrix}, \begin{pmatrix} & & \\ & & y \\ & & \end{pmatrix}, \begin{pmatrix} & & z \\ & & \\ & & \end{pmatrix},$$

and since the operation is commutation of matrices, these are such that $xy - yx = z$. Write this commutator via the Lie bracket, $[x, y] = xy - yx$, and observe that $[y, x] = -[x, y]$ and that all other brackets equal 0.

Suppose that S is a subalgebra of H. It has basis elements, call them a, b, c, and these may be written as linear combinations of x, y, z, thus:

$$\begin{aligned} a &= m_{11}x + m_{12}y + m_{13}z \\ b &= m_{22}y + m_{23}z \\ c &= m_{33}z \end{aligned}$$

for some $m_{ij} \in \mathbb{Z}_p$. If this *is* a subalgebra of H, then the commutators of the elements a, b, c must remain with S. That is, they must be expressible in terms of a, b, c.

From the commutators of x, y, z, it follows that $[a, c] = [b, c] = 0$, and $[a, b] = m_{11}m_{22}[x, y] = m_{11}m_{22}z$. This is expressible in terms of a, b, c if, and only if, the coefficient $m_{11}m_{22}$ is a multiple of m_{33}. So the only constraint on the matrices $M = (m_{ij})$ is that $m_{33}|m_{11}m_{22}$.

Thus, applying this to the formula at the end of appendix A, the zeta function for the Heisenberg algebra is given by

$$\zeta_{H,p}(s) = \left(1 - p^{-1}\right)^{-3} \int\limits_{m_{33}|m_{11}m_{22}} |m_{11}|_p^{s-1} |m_{22}|_p^{s-2} |m_{33}|_p^{s-3} d\nu_p.$$

This is the integral that has been used throughout.

Appendix C

A Full Example

Below is a demonstration of the solver, run in different ways, for the integral

$$\int_{\substack{y|uxz \\ y|uaz \\ y|uex \\ y|uea}} |u|^{s-1}|x|^{2s-4}|y|^{s-4}|z|^{2s-5} d\nu.$$

The file used, therefore, is

```
sqook:~$ cat test
  i := Int |u|^{s-1}|x|^{2s-4}|y|^{s-4}|z|^{2s-5} dv
       y|uxz,
       y|uaz,
       y|uex,
       y|uea.;
```

On the following pages are the solution to this, performed with blow-ups specified or not, and with or without factorising.

Unspecified blow-ups, no factorising.

```
sqook:~$ java Project test

+--------------------------------------------------------------+

            s-1   2s-4   s-4   2s-5
    Int  |u|   |x|    |y|   |z|     dv

   y | u x z
   y | u z a
   y | u x e
   y | u a e

 Enter variables (or press return):

 Finished integral.
 ------------------

 Summing terms  :  [*******************]

+--------------------------------------------------------------+

 i:=(1+X^2*Y^3-X^4*Y^5-X^4*Y^6-X^4*Y^7-X^5*Y^8-X^5*Y^9-X^6*Y^9
 -2*X^6*Y^10-X^7*Y^12+X^8*Y^11+X^8*Y^12+2*X^9*Y^14+2*X^9*Y^15+
 X^9*Y^16+X^10*Y^15+2*X^10*Y^16+2*X^10*Y^17+2*X^11*Y^18+2*X^11
 *Y^19-X^12*Y^18+X^12*Y^19+X^12*Y^20-X^13*Y^20-2*X^13*Y^21-X^1
 4*Y^21-X^14*Y^22-2*X^14*Y^23-X^14*Y^24-2*X^15*Y^24-3*X^15*Y^2
 5-X^15*Y^26-X^16*Y^26-X^16*Y^27+X^17*Y^27-X^17*Y^28-X^17*Y^29
 +X^18*Y^28+X^18*Y^30+X^19*Y^30+2*X^19*Y^31+X^19*Y^32+X^20*Y^3
 3+X^20*Y^34+X^21*Y^35-X^23*Y^37-X^24*Y^40)/((1-X)*(1-X^2*Y^3)
 *(1-X^2*Y^4)*(1-X^3*Y^5)*(1-X^3*Y^6)*(1-X^4*Y^6)*(1-X^4*Y^7)*
 (1-X^5*Y^9)*(1-X^6*Y^10));

+--------------------------------------------------------------+
```

User-specified blow-ups, no factorising.

```
sqook:~$ java Project test

+------------------------------------------------------------+

            s-1   2s-4   s-4   2s-5
    Int  |u|   |x|    |y|   |z|     dv

  y | u x z
  y | u z a
  y | u x e
  y | u a e

 Enter variables (or press return):  x a

 +-----+
 | x|a |
 +-----+

            s-1   2s-3   s-4   2s-5
    Int  |u|   |x|    |y|   |z|     dv

  y | u x z
  y | u x e

 Enter variables (or press return):  e z

 +-----------+
 | x|a : e|z |
 +-----------+

            s-1   2s-3   s-4    2s-5   2s-4
    Int  |u|   |x|    |y|   |z'|    |e|     dv

  y | u x e

 Enter variables (or press return):

 Finished branch.
 ----------------
```

```
+-----------+
| x|a : z|e |
+-----------+

           s-1   2s-3   s-4   2s-4
   Int  |u|   |x|    |y|   |z|     dv

  y | u x z

Enter variables (or press return):

Finished branch.
----------------

+-----------+
| x|a : e=z |
+-----------+

           s-1   2s-3   s-4   2s-4
   Int  |u|   |x|    |y|   |z|     dv

  y | u x z

Enter variables (or press return):

Finished branch.
----------------

+-----+
| a|x |
+-----+

           s-1    2s-4   s-4   2s-5   2s-3
   Int  |u|   |x'|    |y|   |z|    |a|     dv

  y | u z a
  y | u a e

Enter variables (or press return):  e z
```

```
+-----------+
| a|x : e|z |
+-----------+

           s-1   s-4    2s-5   2s-3   2s-4
   Int  |u|   |y|   |z'|    |a|    |e|     dv

  y | u a e

Enter variables (or press return):

Finished branch.
----------------

+-----+
| x=a |
+-----+

           s-1   s-4   2s-5   2s-3
   Int  |u|   |y|   |z|    |a|     dv

  y | u z a
  y | u a e

Enter variables (or press return):

Finished integral.
------------------

Summing terms  :  [*******************]

+---------------------------------------------------------------+

i:=(1-X^2*Y^2-X^4*Y^5-X^4*Y^6-X^4*Y^7-X^5*Y^8+X^6*Y^7+2*X^6*Y
^8+X^6*Y^9+X^7*Y^10+X^7*Y^11-X^8*Y^10+X^8*Y^12+X^8*Y^13-X^9*Y
^13+X^9*Y^15-X^10*Y^14-2*X^10*Y^15-X^11*Y^17-X^11*Y^18+X^12*Y
^17+X^13*Y^20)/((1-X)*(1-X^2*Y^2)*(1-X^2*Y^3)*(1-X^2*Y^3)*(1-
X^2*Y^4)*(1-X^3*Y^5)*(1-X^3*Y^6)*(1-X^4*Y^7));

+---------------------------------------------------------------+
```

Factorising.

```
sqook:~$ java Project -f test

+---------------------------------------------------------------+

            s-1   2s-4   s-4   2s-5
    Int  |u|   |x|    |y|   |z|     dv

   y | u x z
   y | u z a
   y | u x e
   y | u a e

 Enter variables (or press return):

 Finished integral.
 ------------------

 Summing terms  :  [********************]
 Factorising    :  [********************]

+---------------------------------------------------------------+

 i:=(1+X^2*Y^3-X^4*Y^5-X^5*Y^8)/((1-X)*(1-X^2*Y^3)*(1-X^2*Y^4)*
 (1-X^3*Y^5)*(1-X^3*Y^6));

+---------------------------------------------------------------+
```

Gareth Taylor
Corpus Christi
glt1000

Diploma in Computer Science Project Proposal

Solving Certain p-adic Integrals

29th November, 2001

Project Originator: Gareth Taylor

Resources Required: See Project Resources Form

Project Supervisor: Ben Harris

Signature:

Director of Studies: David Greaves

Signature:

Overseers: Chris Hadley and Markus Kuhn

Introduction

This project has its roots in group theory and in recent work on methods for calculating rates of growth of subalgebras of Lie algebras. In the paper [2], the rate of growth is encoded in a sum, called a *zeta function*, and it is shown that this sum is expressible as a p-adic integral. The maths behind this is fairly complicated, but will not be required here – this project will be about solving the p-adic integrals.

Until recently, very few of these integrals had been calculated, but the paper [1] describes a method called *resolution of singularities* to simplify the conditions of a integral by breaking it into smaller parts. This project will be implementing *blow-ups* – a particular type of resolution, described in detail in [3] – to effect these simplifications. As a result, the integrals may be calculated explicitly, without resorting to numerical methods.

The mathematics required will be described informally in the project.

Work to be undertaken

The main part of this project will naturally be writing a program to solve p-adic integrals by the blow-up method. It will be restricted to a certain type of integral – those which form the zeta function of a Lie algebra. These integrals always have comparatively nice conditions, of the form 'monomial divides polynomial', and this form shall be assumed. (For, as with ordinary real integrals, p-adic integrals may be made arbitrarily complicated and there would be no choice but resort to numerical methods.)

The blow-ups consist of various changes of variables, with each blow-up breaking the calculation into two parts, but simplifying the conditions for each part. In order to perform a blow-up, the program will need to analyse the conditions of the integrals to work out which transformations are valid, and then perform them. This will involve an amount of symbolic manipulation, with suitable data structures to store the various terms under consideration.

Eventually, after the appropriate number of blow-ups, the integral in question will have been reduced essentially to a sum of geometric progressions – one of the properties of working with p-adic numbers. The methods for

summing, rearranging and factorizing large expressions of this form are already implemented by algebra packages such as Maple, so these will not be a necessary part of this project. Indeed, producing output in a form acceptable as input to Maple would be not only sufficient but useful.

Variants and Extensions

Frequently, there will be a choice of which blow-up to perform, and the order in which they are done will often alter the length of the calculation; a wise transformation early on may save several later. This is sometimes predictable behaviour, and incorporating this would be a useful addition to the program.

When performing these sorts of calculations by hand, it is sometimes possible to see that certain transformations will be more efficient in producing the answer, usually by performing a more complicated change of variables than a simple blow-up. It may be possible to include some of these shortcuts, although it would require a more careful analysis of the conditions at each stage.

To begin with, the input will be assumed to be valid and correctly formatted. Later, this should be changed to include some checking and so on.

Finally, there would be some benefit in being able to derive the conditions for the integrals from their associated algebras. The structure and relations of the algebra give rise to the conditions, and it would be possible to allow them to be calculated by giving details of the algebra instead. This would mainly be a labour-saving device, but would be a nice addition, if time permits.

References

[1] M.P.F. du Sautoy and F.J. Grunewald, *Analytic properties of zeta functions and subgroup growth*, Annals of Math. **152** (2000), 793–833

[2] F.J. Grunewald, D. Segal, G.C. Smith, *Subgroups of finite index in nilpotent groups*, Invent. Math. **93** (1988), 185–223

[3] G.L. Taylor, *Zeta functions of algebras and resolution of singularities*, PhD thesis, University of Cambridge, 2001

Resources

No resources should be required for this project beyond my present allocation on the PWF and Thor.

Starting Point

The mathematics behind this project was developed as part of my PhD, but for the purposes of this project, the vast majority will not be required. A basic understanding of p-adic numbers is required, and there will be appropriate explanations included in the project report.

Timetable and Milestones

15th – 28th November (2 weeks)

Discussion with Overseers and Director of Studies, and allocation of a Project Supervisor. Writing Project Proposal. *Milestones:* a complete Project Proposal complete with realistic timetable and approval from Overseers.

29th November – 2nd January (5 weeks)

December will have to consist of reading due to a lack of computing resources over the Christmas vacation. This will include preliminary study of ML, to see whether it would be a more suitable choice of language for the project than Java.

3rd – 16th January (2 weeks)

Practical testing ML versus Java and making a more informed choice about which is more appropriate. Implementing small programs based on parts of the project to judge suitability. A draft version of the Introduction chapter can be written. *Milestones:* a sensible choice of the language to be used, and some helpful small starting bits of code written during the choosing.

17th January – 13th February (4 weeks)

Begin implementing data structures to encode the monomial and polyno-

mial terms. Write the parts of the program which will compare and reduce the monomials. *Milestones:* a program capable of performing blow-ups on monomial conditions.

14th February – 6th March (3 weeks)

Implementation of the simpler forms of integral – those whose conditions are solely monomials. This will form the basis of the later, more complicated calculations. Also, begin the Preparation chapter of the Dissertation. *Milestones:* having a program to solve monomial integrals, a suitable test case being the Heisenberg Lie Algebra.

7th March – 3rd April (4 weeks)

Begin the more complicated calculations! In particular, writing the parts of the program which compare and reduce the polynomial terms. *Milestones:* a program capable of performing blow-ups on polynomial conditions – in particular to reduce them to monomials.

4th April – 1st May (4 weeks)

Implementation of more complicated forms of integral – those which have polynomial conditions. For now, success should be restricted to linear or quadratic polynomials. Writing the Implementation chapter, covering the monomial work. *Milestones:* having a program to solve smaller polynomial integrals, a suitable test case being $\mathfrak{sl}_2(\mathbb{Z})$ or $\mathfrak{o}_3(\mathbb{Z})$.

2nd – 15th May (2 weeks)

Testing of the program to this point, involving working out known results, noting any problems that arise and attempting to resolve them. *Milestones:* the correct evaluation of a number of suitable integrals, which will be detailed in the dissertation.

16th – 29th May (2 weeks)

Further testing, plus attempting to solve new integrals, for which the results are not yet known. Project work most likely restricted to testing at this stage, to allow for exam revision.

10th – 26th June (2 weeks)

The majority of the calculating part of the implementation should be be complete, and that chapter of the Dissertation should be essentially complete. Writing the evaluation chapter should begin. The inclusion to the program of some of the additional simplification methods. *Milestones:* a good set of results and test-cases, a large amount of the Dissertation complete and proof-read by friends and Supervisor.

27th June – 17th July (3 weeks)

If time allows, investigate some of the variants and extensions mentioned above. General tidying up of program and results, resolving any final problems.

18th – 31st July (2 weeks)

Finish Dissertation, review project, proof-read. And finally, submission of the Dissertation.