The MZV Datamine

The goal of this lecture is to study and understand the Multiple Zeta Value (MZV) datamine and how it was created. The mathematical details and many of the results are described in the paper

J. Blümlein, D.J. Broadhurst and J.A.M. Vermaseren, "The Multiple Zeta Value Data Mine", Comput.Phys.Commun. 181 (2010) 582-625, arXiv:0907.2557.

Of course we cannot go into all details. This paper touches on many things that are not yet fully understood. But MZV's, the related Euler sums and the related harmonic sums and harmonic polylogarithms have become rather important in some branches of physics (and mathematics?) these days, so it is nice if you know some ways to deal with them.
The program we will be studying here is one that generates relations for all MZV's of a given weight, expressing them in terms of a minimal basis. Currently the only known way to do this, is to derive many relations in terms of these objects and then solving this (linear) set of equations. It can run out of hand easily.

The first thing to do is to make a plan. We first make an inventory of what we need:

- We need a good notation.
- We need an efficient algorithm.
- We may have to optimize the program heavily.

For the part about optimization the selection of the language is sometimes important. Because we want to take this project to as high a weight as possible FORM, and in particular TFORM is a good choice.
A good notation is as important as a good algorithm, specially when the code will need to be optimized. This is sometimes forgotten. Hence we will start with some basics and establish a notation that seems to work well.

The harmonic series is defined by

$$\begin{aligned} S_m(n) &= \sum_{i=1}^{n} \frac{1}{i^m} \\ S_{-m}(n) &= \sum_{i=1}^{n} \frac{(-1)^i}{i^m} \\ S_{m,j_1,\cdots,j_p}(n) &= \sum_{i=1}^{n} \frac{1}{i^m} S_{j_1,\cdots,j_p}(i) \\ S_{-m,j_1,\cdots,j_p}(n) &= \sum_{i=1}^{n} \frac{(-1)^i}{i^m} S_{j_1,\cdots,j_p}(i) \end{aligned}$$

in which $m, j_1, \cdots, j_p > 0$. The m and the j_i are referred to as the indexes of the harmonic series. Hence

$$S_{1,-5,3}(n) = \sum_{i=1}^{n} \frac{1}{i} \sum_{j=1}^{i} \frac{(-1)^j}{j^5} \sum_{k=1}^{j} \frac{1}{k^3}$$

In the literature the alternating sums are usually indicated by a bar over the index. Because this is hard to achieve in a computer algebra system, we prefer to work with the sign.

Mathematicians prefer a slightly different definition which comes down from a definition for infinite sums. They define:

$$\begin{aligned}
Z_m(n) &= \sum_{i=1}^{n} \frac{1}{i^m} \\
Z_{-m}(n) &= \sum_{i=1}^{n} \frac{(-1)^i}{i^m} \\
Z_{m,j_1,\cdots,j_p}(n) &= \sum_{i=1}^{n} \frac{1}{i^m} Z_{j_1,\cdots,j_p}(i-1) \\
Z_{-m,j_1,\cdots,j_p}(n) &= \sum_{i=1}^{n} \frac{(-1)^i}{i^m} Z_{j_1,\cdots,j_p}(i-1)
\end{aligned}$$

This notation is related to the definition of Multiple Zeta Values which is by

$$\zeta_{\vec{a}} = \lim_{N\to\infty} Z_{\vec{a}}(N)$$

in which $\vec{a}$ is a string of positive indexes.
Both definitions have their advantages. In physics, when doing calculations involving Mellin moments usually the S notation is used.

Because of limitations in time from now on we will ignore the negative indexes and assume that all indexes are positive. The infinite sums that involve also negative indexes are called Euler sums. The datamine also has tables and programs for those.
The notation we select is as in

$$S_{i_1,\cdots,i_m}(n) \to \texttt{S(R(i1,...,im),n)}.$$

The reason we work with the extra function R will become clear soon. Let us first play around a bit. We start with a little program to evaluate a finite sum.

```
#define MAX "5"
CFunction S,R;
Symbols m,n,j;

#do N = 1,`MAX'
L   F`N' = S(R(1),`N');
#enddo
id  S(R(m?),n?pos_) = sum_(j,1,n,1/j^m);
Print;
.end
```

```
F1 =
    1;

F2 =
    3/2;

F3 =
    11/6;

F4 =
    25/12;

F5 =
    137/60;
```

There is nothing special about this program. It becomes more interesting when we allow more indexes:

```
 #define MAX "5"
 CFunction S,R;
 Symbols m,n,j;

 #do N = 1,`MAX'
 L   F`N' = S(R(2,1,3),`N');
 #enddo
 repeat id   S(R(m?,?a),n?pos_) = sum_(j,1,n,S(R(?a),j)/j^m);
 id  S(R,n?) = 1;
 Print;
 .end

F1 =
   1;

F2 =
   89/64;
```

```
F3 =
   74989/46656;

F4 =
   5218129/2985984;

F5 =
   86164141609/46656000000;
```

As you see, the fractions become quickly more complicated. Things become worse when we also look at the statistics:

```
CFunction S,R;
Symbols m,n,j;
Format 64;

#do N = 20,20
L   F‘N’ = S(R(2,1,3,1),‘N’);
#enddo
repeat id   S(R(m?,?a),n?pos_) = sum_(j,1,n,S(R(?a),j)/j^m);
```

```
    id  S(R,n?) = 1;
    Print;
    .end

Time =       0.03 sec    Generated terms =       8855
             F20         Terms in output =          1
                         Bytes used      =         68

   F20 =
      858533392742942674937218914704563861240679752868897352107\
      33/37049612808521283635526987908472903984093735010959360\
      00000;
```

To work out the nested sums we generate many terms. In this case $\binom{20+3}{4} = 8855$. We want to avoid this. The standard way we learned till now was to use a #do loop with a .sort inside, do each time one sum, and stay inside the loop as long as we still have to evaluate sums. In this case there is a better method:

```
    CFunction S,R,aux;
```

```
    Symbols m,n,j;
    Format 64;

    #do N = 40,40
    L   F‘N’ = S(R(2,1,3,1),‘N’);
    #enddo
    id  S(R(m?,?a),n?pos_) = aux(S(R(m,?a),n));
    repeat;
      Argument aux;
        id  S(R(m?,?a),n?pos_) = sum_(j,1,n,S(R(?a),j)/j^m);
        id  S(R,n?) = 1;
      EndArgument;
    endrepeat;
    id  aux(n?) = n;
    Print;
    .end

Time =       0.01 sec    Generated terms =          1
```

```
F40          Terms in output =          1
             Bytes used      =        108

F40 =
   30234518788019613338344296592191646288135977835865839249 2\
   13679713030401885440578569691363470369160775800525858 7/
   12429601909674710525976226226490824103824807852113660043 2\
   30232864915471226916474623678688190791680000000000000 0;
```

What happens here is that we put the sum inside an auxiliary function aux and now we substitute one sum at a time. Each time the program runs into the endargument statement the argument is sorted and the whole is put inside the function. The first time this may not do very much, but the second time there are already great savings. Eventually there will be only a number left inside the function aux.

Why did we not use this method before? The problem with this method is that if the sorted function argument becomes very complicated it may be too big to be stored inside a single term. FORM has a maximum for the space that each term may occupy. At the beginning of the program, when memory is allocated, the size of many buffers is related to this maximum size. Hence, putting it extremely large makes that FORM, and specially TFORM, will try to

allocate very much memory. This could fail....
Anyway, for the evaluation of the sums it seems to work well. We could make this into a procedure inside a library as in

```
CFunction evalSaux;
Symbols evalSj,evalSm,evalSn;
*
#procedure evalS(S,R,par)
*
*  Procedure for the evaluation of harmonic sums with postive integer
*  argument. It handles both the S-sum and the Z-sum definitions.
*  Call with
*       #call evalS(SumName,RName,S)   for S-sums
*       #call evalS(SumName,RName,Z)   for Z-sums
*  The SumName and RName should be declared as functions in the
*  calling program (either commuting or non-commuting).
*
id  `S'(`R'(evalSm?,?a),evalSn?pos_) =
                    evalSaux(`S'(`R'(evalSm,?a),evalSn));
```

```
id  `S'(`R'(evalSm?,?a),0) = 0;
repeat;
  Argument evalSaux;
    #switch `par'
    #case S
      id  `S'(`R'(evalSm?pos_,?a),evalSn?pos_) =
             sum_(evalSj,1,evalSn,`S'(`R'(?a),evalSj)/evalSj^evalSm);
      id  `S'(`R',evalSn?) = 1;
    #break
    #case Z
      id  `S'(`R'(evalSm?pos_,?a),evalSn?pos_) =
           sum_(evalSj,1,evalSn,`S'(`R'(?a),evalSj-1)/evalSj^evalSm);
      id  `S'(`R',evalSn?) = 1;
      id  `S'(`R'(?a),0) = 0;
    #break
    #default
      #write "Unrecognized last argument in procedure evalS: `par'"
      #write "Argument should be either S or Z"
```

```
        #terminate
      #break
      #endswitch
    EndArgument;
  endrepeat;
  id  evalSaux(evalSn?) = evalSn;
#endprocedure
```

We put this inside a file named sumlib1.h and now we can use it as in

```
#include- sumlib1.h
CFunction S1,Z1,R,acc;
Symbols m,n,j;
Format 64;

L   F2 = S1(R(2),5);
L   G2 = Z1(R(2),5);
L   F21 = S1(R(2,1),5);
L   G21 = Z1(R(2,1),5);
#call evalS(S1,R,S)
```

```
 #call evalS(Z1,R,Z)
 Print;
 .end

F2 =
   5269/3600;

G2 =
   5269/3600;

F21 =
   388853/216000;

G21 =
   59/96;
```

This procedure can handle both the physicists definition and the mathematicians definition, depending on the last parameter.
Thus far we did not really need the function R. Hence let us solve a little problem that will

make clear why we need it. Let us try to do the following sum:

$$F = \sum_{j=1}^{N} \frac{1}{(j+3)^3} S_{2,1}(j+1)$$

The program would run something like this:

```
#include- sumlib1.h
CFunction sum,den,S,R;
Symbols j,N,m1,m2,x1,x2;

Local F = sum(j,1,N)*den(j+3)^3*S(R(2,1),j+1);
Print;
.sort

F =
   sum(j,1,N)*den(3 + j)^3*S(R(2,1),1 + j);

SplitArg,(j),den,S;
Print;
.sort
```

```
F =
   sum(j,1,N)*den(3,j)^3*S(R(2,1),1,j);
```

To do the sum we have to make sure that the denominators and the sum have the same arguments. Hence we have to synchronize them. We do this with the splitarg statement (we have seen this before). The SplitArg statement has the property that each term in the argument becomes a single argument. This means that `(j)` $\rightarrow$ `(j)` and not `(0,j)`. The result is that if we would not protect the indexes inside the separate function R, it might become hard to keep the indexes apart from pieces of the argument (it could be done at the cost of expensive pattern matching at a later stage).
Next we synchronize the arguments. Try to figure out what the repeat loop does exactly.

```
repeat;
  id  S(R(m1?,?a),x1?!{x2?},j)*den(x2?!{x1?},j) =
        +theta_(x1-x2)*(S(R(m1,?a),x1-1,j)
           +S(R(?a),x1,j)*den(x1,j)^m1)*den(x2,j)
        +theta_(x2-x1)*(S(R(m1,?a),x1+1,j)
           -S(R(?a),x1+1,j)*den(x1+1,j)^m1)*den(x2,j);
  repeat id den(x1?!{x2?},j)*den(x2?!{x1?},j) =
```

```
          (den(x1,j)-den(x2,j))/(x2-x1);
 endrepeat;
 Print +s;
 .sort

F =
     - sum(j,1,N)*den(2,j)^2*S(R(1),2,j)
     + 3*sum(j,1,N)*den(2,j)*S(R(1),2,j)
     - sum(j,1,N)*den(3,j)^5*S(R(1),3,j)
     + sum(j,1,N)*den(3,j)^4*S(R,3,j)
     - sum(j,1,N)*den(3,j)^3*S(R(1),3,j)
     + sum(j,1,N)*den(3,j)^3*S(R(2,1),3,j)
     + 2*sum(j,1,N)*den(3,j)^3*S(R,3,j)
     - 2*sum(j,1,N)*den(3,j)^2*S(R(1),3,j)
     + 3*sum(j,1,N)*den(3,j)^2*S(R,3,j)
     - 3*sum(j,1,N)*den(3,j)*S(R(1),3,j)
   ;
```

At this point we can do the sum:

```
 id  den(x1?,j) = den(1,x1,j);
 repeat id den(m1?,x1?,j)*den(m2?,x1?,j) = den(m1+m2,x1,j);
 id  sum(j,1,N)*den(m1?,x1?,j)*S(R(?a),x1?,j) =
             S(R(m1,?a),N+x1)-S(R(m1,?a),x1);
 Print +s;
 .sort

F =
    + 3*S(R(1,1),2 + N)
    - 3*S(R(1,1),3 + N)
    - 3*S(R(1,1),2)
    + 3*S(R(1,1),3)
    + 3*S(R(2),3 + N)
    - 3*S(R(2),3)
    - S(R(2,1),2 + N)
    - 2*S(R(2,1),3 + N)
    + S(R(2,1),2)
```

```
   + 2*S(R(2,1),3)
   + 2*S(R(3),3 + N)
   - 2*S(R(3),3)
   - S(R(3,1),3 + N)
   + S(R(3,1),3)
   + S(R(3,2,1),3 + N)
   - S(R(3,2,1),3)
   + S(R(4),3 + N)
   - S(R(4),3)
   - S(R(5,1),3 + N)
   + S(R(5,1),3)
  ;
```

We notice many sums with an integer argument. Those sums can be done with the procedure we made before. Hence:

```
#call evalS(S,R,S)
Print +s;
.end
```

```
F =
     - 1/27
     + 3*S(R(1,1),2 + N)
     - 3*S(R(1,1),3 + N)
     + 3*S(R(2),3 + N)
     - S(R(2,1),2 + N)
     - 2*S(R(2,1),3 + N)
     + 2*S(R(3),3 + N)
     - S(R(3,1),3 + N)
     + S(R(3,2,1),3 + N)
     + S(R(4),3 + N)
     - S(R(5,1),3 + N)
   ;
```

How do we know that this answer is correct? We could put some numbers:

```
Drop;
Local F1 = F*replace_(N,1);
```

```
 Local F2 = F*replace_(N,2);
 Local F3 = F*replace_(N,3);
 Local F4 = F*replace_(N,4);
 Local F5 = F*replace_(N,5);
 #call evalS(S,R,S)
 Print;
 .end

F1 =
   11/512;

F2 =
   58949/1728000;

F3 =
   490187/11664000;

F4 =
```

```
   189132203/4000752000;

F5 =
   52155148301/1024192512000;
```

And now for a check we can put the same numbers in the original formula:

```
#include- sumlib1.h
CFunction sum,den,S,R;
Symbols j,N,m1,m2,x1,x2;

#do N = 1,5
Local F‘N’ = sum_(j,1,‘N’,den(j+3)^3*S(R(2,1),j+1));
#enddo
Print;
id  den(x1?) = 1/x1;
#call evalS(S,R,S)
Print;
.end
```

```
F1 =
   11/512;

F2 =
   58949/1728000;

F3 =
   490187/11664000;

F4 =
   189132203/4000752000;

F5 =
   52155148301/1024192512000;
```

and as you can see, the answers are identical. It shows you one of the very powerful properties of finite sums: at any moment you can substitute numbers and obtain exact evaluations. This aids greatly in debugging a program.

The next thing is that the sums obey some kind of shuffle algebra. This is based on the property of sums that

$$\begin{aligned}\sum_{i=1}^{N}\sum_{j=1}^{N} f(i,j) &= +\sum_{i=1}^{N}\sum_{j=1}^{i} f(i,j) + \sum_{j=1}^{N}\sum_{i=1}^{j} f(i,j) - \sum_{i=1}^{N} f(i,i) \\ &= +\sum_{i=1}^{N}\sum_{j=1}^{i-1} f(i,j) + \sum_{j=1}^{N}\sum_{i=1}^{j-1} f(i,j) + \sum_{i=1}^{N} f(i,i)\end{aligned}$$

The first identity being the one for the S-sums and the second for the Z-sums. In FORM this can be programmed in a short procedure as in

```
repeat;
 id S(R(?a),x?)*S(R(?b),x?) = S(R,R(?a),R(?b),x);
 repeat id S(R(?a),R(x1?,?b),R(x2?,?c),x?)=
        +S(R(?a,x1),R(?b),R(x2,?c),x)
        +S(R(?a,x2),R(x1,?b),R(?c),x)
        'SIGN'S(R(?a,x1+x2),R(?b),R(?c),x);
 id S(R(?a),R,R(?c),x?) = S(R(?a,?c),x);
 id S(R(?a),R(?b),R,x?) = S(R(?a,?b),x);
endrepeat;
```

in which the preprocessor variable SIGN is - for S-sums and + for Z-sums. Because of this

extra term, the algebra is not quite a shuffle algebra. There is extra stuff. Hence Broadhurst has labeled it a stuffle algebra. Considering that such sums are becoming more and more common in several types of physics, FORM has been equiped with a special statement for this: the stuffle statement. We would use

```
Stuffle,S-;
Stuffle,Z+;
```

in which S or Z is the name of the function whose arguments have to be stuffled, and the - or + indicates which type of sum is involved. Because of the extra function R inside the S we would use

```
id,once,S(R(?a),x?)*S(R(?b),x?) = R(?a)*R(?b)*RR(x);
Stuffle,R-;
id R(?a)*RR(x?) = S(R(?a),x);
```

When we study sums in infinity, we will drop the function R and the argument of the sum. In that case things become rather straightforward.

The summer library contains many procedures that can handle several categories of complicated nested sums. We will skip that here, because we do not have time for it.

The sum of the absolute value of the indexes of a sum is called its weight, and the number of indexes its depth. How many sums are there?

An alternative notation for the index field of the sums is that we represent a positive index n by $n-1$ zeroes followed by a single 1. In this notation it is rather easy to see how many sums there are. The last index should always be a one. The number of indexes in this notation is the weight. Hence there are 2^{w-1} different sums for a given weight, because each index (except for the last one) can be either 0 or 1. For alternating sums a negative index $-n$ will be $n-1$ zeroes followed by -1. Hence there are three possibilities for each index, except for the last one, for which there are two possibilities. Hence there are $2\ 3^{w-1}$ different alternating sums for a given weight.

Next, there are the harmonic polylogarithms, which we will also call H-functions. We consider the alphabets

$$\begin{aligned} h &= \{0, 1, -1\} \quad \text{and} \\ H &= \{1/x, 1/(1-x), 1/(1+x)\} \ , \end{aligned} \tag{1}$$

which define the elements of the index set of the harmonic polylogarithms and the functions in the iterated integrals, respectively. Let $\vec{a} = \{m_1, \ldots, m_k\}, \quad m_i, b \ \in \ h, \quad k \geq 1$, then

$$\begin{aligned} H_{b,\vec{a}}(x) &= \int_0^x dz f_b(z) H_{\vec{a}}(z) \\ f_0(z) &= 1/z \\ f_1(z) &= 1/(1-z) \\ f_{-1}(z) &= 1/(1+z) \\ H_0(x) &= \quad \log(x) \\ H_1(x) &= -\log(1-x) \\ H_{-1}(x) &= \quad \log(1+x) \ . \end{aligned}$$

The number of indexes is again called the weight and the number of nonzero indexes is called the depth. We see that the notation here is similar to the alternative notation for the sums above. This is why we call the original notation for the sums the sum-notation, and

the notation with the 0,1,-1 the integral-notation. It is not very complicated to write FORM code that can convert from one notation to the other. This will be necessary, because it turns out that the sums to infinity and the H-functions at unity are all related and can be readily transformed into each other. For some applications it is most convenient to work with one set of objects and for others the other objects may be more useful. For reasons being explained later our computer programs work mostly with H-functions at unity. As it turns out, when we go to high weights (like more than 20), the conversion of one notation into the other can become rather time-consuming, hence we have special commands for this in FORM. These are part of a larger family of commands, designed to manipulate argument fields of functions. This is the transform statement, which currently has already 10 options. We will see it in the code later.
The H-functions can be represented in a similar way as the sums:

```
H(R(0,1,1,0,0,-1),x)
```

and the harmpol library contains a number of useful procedures, including Mellin transforms of H-functions into S-sums and inverse Mellin transforms to go from S-sums to H-functions. This is however beyond the scope of this lecture.
The important thing is that if the indexes are all zero or positive the H-functions in one are equal to the Z-sums in infinity and are equal to the corresponding multiple zeta values. When

negative indexes are involved there can be some sign differences, but it is possible to write a simple procedure for this. It is also not complicated to convert the S-sums into Z-sums et vise versa.
Broadhurst has shown that when perturbation theory will reach a sufficient number of loops we will obtain infinite sums that are not just alternating, but instead of $(-1)^i$ we will have a^i with a a sixth root of unity (-1 being a second root of unity). In that case there would be $6\ 7^{w-1}$ multiple zeta values of a given weight. This explodes rather rapidly. However FORM is ready for this.
To convert from one notation to the other we use the transform statement as in:

```
CFunction
CFunction H;
Symbols x;

Local F = H(3,-5,2);
Repeat id H(?a,x?!{-1,0,1},?b) = H(?a,0,x-sig_(x),?b);
Print;
.sort
```

```
F =
   H(0,0,1,0,0,0,0,-1,0,1);

 Repeat id H(?a,0,x?!{0,},?b) = H(?a,x+sig_(x),?b);
 Print;
 .sort

F =
   H(3,-5,2);

 Transform,H,tointegralnotation(1,last);
 Print;
 .sort

F =
   H(0,0,1,0,0,0,0,-1,0,1);

 Transform,H,tosumnotation(1,last);
```

```
 Print;
 .sort

F =
   H(3,-5,2);

 Symbol a#=5,j;
 Drop;
 Local F = H(3*a,5,2*a)+sum_(j,1,12,a^j);
 Transform,H,tointegralnotation(1,last);
 Print;
 .end

F =
   2 + 3*a + 3*a^2 + 2*a^3 + 2*a^4 + H(0,0,a,0,0,0,0,1,0,a);
```

You see this working for positive and negative indexes. The final module shows that for any root of unity (in this case a fifth root of unity) the transform statement will work (only with versions of FORM after 18-feb-2014, as there was still a bug, because apparently nobody has

used it yet). The first modules show how one could convert from one notation to the other by 'external' means. This is however far less efficient, because it involves much pattern matching.

The H-functions obey a proper shuffle algebra, when written in integral notation. To facilitate this, FORM has also a shuffle statement. Because there can be many adjacent indexes that are identical, it is important that this statement has a lot of combinatorics built in, even though it still can generate identical terms. Yet the shuffle statement is far superior to any external implementation.

```
CFunction S,R,H,R1,R2;
Symbols N,x;
Local F = S(R(1,5),N)*S(R(2,3),N);
Local G = H(R(1,0,0,0,0,1),x)*H(R(0,1,0,0,1),x);
id  S(R(?a),x?)*S(R(?b),x?) = R1(?a)*R1(?b)*R2(x);
Stuffle R1-;
id  R1(?a)*R2(x?) = S(R(?a),x);
id  H(R(?a),x?)*H(R(?b),x?) = R1(?a)*R1(?b)*R2(x);
Shuffle R1;
id  R1(?a)*R2(x?) = H(R(?a),x);
.end

Time =       0.00 sec    Generated terms =         13
```

```
                F          Terms in output =          13
                           Bytes used      =        1108

Time =        0.00 sec     Generated terms =          62
                G          Terms in output =          39
                           Bytes used      =        3052
```

At this point we come to the essence of the datamine. If the sums obey a stuffle relation that expresses the product of two lower weight sums into a sum over terms with a weight that is the sum of those lower weights, we could try to write down all such stuffle equations and then invert that set of equations to express as many higher weight sums as possible into products of lower weight sums and as few remaining sums as possible of the original weight.

```
CFunction E,ZZ,Z;
Symbol Sinf;
L   Z21 = E(1)*E(1);
L   Z31 = E(2)*E(1);
L   Z32 = E(1,1)*E(1);
L   Z41 = E(3)*E(1);
L   Z42 = E(2,1)*E(1);
L   Z43 = E(1,2)*E(1);
L   Z44 = E(1,1,1)*E(1);
L   Z45 = E(2)*E(2);
L   Z46 = E(1,1)*E(2);
L   Z47 = E(1,1)*E(1,1);
id  E(?a)*E(?b) = ZZ(?a)*ZZ(?b)-Z(?a)*Z(?b);
```

```
 Stuffle Z+;
 id  ZZ(?a) = Z(?a);
 id  Z(1) = Sinf;
 Format nospaces;
 Print +f;
 .end

Z21=
   Sinf^2-2*Z(1,1)-Z(2);

Z31=
   -Z(1,2)+Z(2)*Sinf-Z(2,1)-Z(3);

Z32=
   Z(1,1)*Sinf-3*Z(1,1,1)-Z(1,2)-Z(2,1);

Z41=
   -Z(1,3)+Z(3)*Sinf-Z(3,1)-Z(4);
```

```
Z42=
   -Z(1,2,1)+Z(2,1)*Sinf-2*Z(2,1,1)-Z(2,2)-Z(3,1);

Z43=
   -2*Z(1,1,2)+Z(1,2)*Sinf-Z(1,2,1)-Z(1,3)-Z(2,2);

Z44=
   Z(1,1,1)*Sinf-4*Z(1,1,1,1)-Z(1,1,2)-Z(1,2,1)-Z(2,1,1);

Z45=
   Z(2)^2-2*Z(2,2)-Z(4);

Z46=
   Z(1,1)*Z(2)-Z(1,1,2)-Z(1,2,1)-Z(1,3)-Z(2,1,1)-Z(3,1);

Z47=
   Z(1,1)^2-6*Z(1,1,1,1)-2*Z(1,1,2)-2*Z(1,2,1)-2*Z(2,1,1)-Z(2,2);
```

The symbol `Sinf` stands for `Z(1)` and is the basic divergence. It is such a soft (logarithmic) divergence that in many relations it can be used as a symbol and be eliminated from the equations. In some cases one has to be careful though. This holds in particular when converting from H-functions in one to Z-sums in infinity. Only if a single divergence is present (no more than a single leading one in the index field) this does not need correction terms. For the shuffle relation of the H-functions no correction terms are needed when only a single H is divergent. This is because the shuffle relation is related to the 'rule of the triangle' in the paper about harmonic sums. In our case we will only use stuffle and shuffle relations with at most one divergent object, and only with a single power of `Sinf`. This way we avoid all problems. We will see though that such equations are needed.

If one has to worry only about either the stuffle relations or the shuffle equations, many things are known. A basis in the higher weights can be formed by so-called Lyndon words of indexes. **skip that here.** The real interest here is that the MZV's, being both sums in infinity and HPL's in one, obey both the stuffle algebra and the shuffle algebra, the first in sum-notation and the second in integral-notation.

If we apply the shuffle relations to the expressions above that qualify we obtain:

```
CFunction E,ZZ,Z;
Symbol Sinf;
L   Z31 = E(2)*E(1);
L   Z41 = E(3)*E(1);
L   Z42 = E(2,1)*E(1);
L   Z45 = E(2)*E(2);
id  E(?a)*E(?b) = ZZ(?a)*ZZ(?b)-Z(?a)*Z(?b);
Transform,Z,ToIntegralNotation(1,last);
Shuffle Z;
Transform,Z,ToSumNotation(1,last);
id  ZZ(?a) = Z(?a);
id  Z(1) = Sinf;
Format nospaces;
Print +f;
.end

Z31=
```

```
   -Z(1,2)+Z(2)*Sinf-2*Z(2,1);

Z41=
   -Z(1,3)-Z(2,2)+Z(3)*Sinf-2*Z(3,1);

Z42=
   -Z(1,2,1)+Z(2,1)*Sinf-3*Z(2,1,1);

Z45=
   Z(2)^2-2*Z(2,2)-4*Z(3,1);
```

If we take the two Z31 equations we see

```
0 =   -Z(1,2)+Z(2)*Sinf-Z(2,1)-Z(3);
0 =   -Z(1,2)+Z(2)*Sinf-2*Z(2,1);
```

By subtracting them we can eliminate the divergent pieces. It can be shown by inserting proper limits that this is a safe procedure and hence

```
0 = Z(2,1)-Z(3);
```

and we obtain the famous Euler relation $\zeta_{2,1} = \zeta_3$.

Now we know the problem we can fix our strategy.

The first thing that might come to mind is to generate all equations that can be used and construct a big matrix. Then use standard solving techniques to solve this problem. People have tried this, but because the matrix is rather sparse, this uses way too much space. It also becomes needlessly slow, because most of the time the program is manipulating zeroes.

We will use a method that has a single (big) expression that contains all knowledge we have at a given time. This expression starts as (for weight 4)

```
+E(0,0,0,1)*(H(0,0,0,1))
+E(0,0,1,1)*(H(0,0,1,1))
+E(0,1,0,1)*(H(0,1,0,1))
```

and we call it the 'master expression'. The contents of the brackets are what we know at a given moment about the object outside the brackets. Next we generate an equation. We start with the finite stuffles. Of these there is only one:

```
H(2)*H(2) = 2*H(2,2)+H(4)
```

or

```
H(0,1)*H(0,1) = 2*H(0,1,0,1)+H(0,0,0,1)
```

We start with inserting information about lower weight objects. In this case there is only

$$H_{0,1} = \zeta_2$$

and we call ζ_2 `z2` in the program.

We can use the above equation to eliminate `H(0,1,0,1)` in the master expression which becomes

```
+E(0,0,0,1)*(H(0,0,0,1))
+E(0,0,1,1)*(H(0,0,1,1))
+E(0,1,0,1)*(-H(0,0,0,1)/2+z2^2/2)
```

Next we generate the corresponding shuffle equation:

```
0 = z2^2-4*H(0,0,1,1)-2*H(0,1,0,1)
```

The 'magic' step is now to substitute the H-functions in the rhs by the contents of the corresponding E bracket:

```
0 = z2^2-4*H(0,0,1,1)-2*(-H(0,0,0,1)/2+z2^2/2)
  = -4*H(0,0,1,1)+H(0,0,0,1)
```

Now we can substitute `H(0,0,1,1)` in the master expression to obtain

```
+E(0,0,0,1)*(H(0,0,0,1))
+E(0,0,1,1)*(1/4*H(0,0,0,1))
+E(0,1,0,1)*(-H(0,0,0,1)/2+z2^2/2)
```

At this point we are not finished yet, because if we combine the two (divergent) Z41 equations we saw above we have

```
0 = -Z(1,3)-Z(2,2)+Z(3)*Sinf-2*Z(3,1)
0 = -Z(1,3)+Z(3)*Sinf-Z(3,1)-Z(4)
```

and by subtraction the divergences vanish to give the equation

```
0 = -H(0,1,0,1)-H(0,0,1,1)+H(0,0,0,1)
```

Again we substitute the contents of the corresponding brackets in the master expression to obtain the relation

```
0 = -(-H(0,0,0,1)/2+z2^2/2)-(1/4*H(0,0,0,1))+H(0,0,0,1)
  =  5/4*H(0,0,0,1)-z2^2/2
```

and we substitute

```
id  H(0,0,0,1) = 2/5*z2^2;
```

into the master expression to reach the final expression

```
+E(0,0,0,1)*(2/5*z2^2)
+E(0,0,1,1)*(1/10*z2^2)
+E(0,1,0,1)*(3/10*z2^2)
```

Now we can read off:

```
z4 = Z(4) = H(0,0,0,1) = 2/5*z2^2
```

which is a well known relation.
There remains one little point. Why did we not have `E(0,1,1,1)` in the master expression? As it turns out, there is a duality relation for the integrals in which reverting the order of the indexes and exchanging zeroes and ones gives an integral with the same value. This saves us half the number of objects to compute for odd weights and almost half for even weights (there are objects like $H_{0,1,0,1}$ which are selfdual).

Next we consider that storing objects like

```
H(0,1,0,0,1,0,0,1,1,0,0,0,0,1,0,1,0,0,1,0,0,1)
```

costs much space and makes looking up the brackets also not easier, while actually this argument looks like a large binary number. This we can encode of course at the cost of much pattern matching, but again the transform statement is eminently suitable for this:

```
CFunction H;
Local F = H(0,1,0,0,1,0,0,1,1,0,0,0,0,1,0,1,0,0,1,0,0,1);
Transform,H,encode(1,last):base=2;
Print;
.sort

F =
   H(1204553);

Transform,H,decode(1,22):base=2;
Print;
.end
```

```
F =
   H(0,1,0,0,1,0,0,1,1,0,0,0,0,1,0,1,0,0,1,0,0,1);
```

Here we have to specify how many binary digits we want. For reasons of efficiency and ordering of the terms we prefer to exchange the zeroes and the ones before we use this encoding. Now comes one of the nice features of the transform statement. We can do this all in the same step:

```
CFunction H;
Local F = H(0,1,0,0,1,0,0,1,1,0,0,0,0,1,0,1,0,0,1,0,0,1);
Transform,H,replace(1,last)=(0,1,1,0)
           ,encode(1,last):base=2;
Print;
.sort

F =
   H(2989750);

Transform,H,decode(1,22):base=2
```

```
            ,replace(1,last)=(0,1,1,0);
 Print;
 .end

F =
   H(0,1,0,0,1,0,0,1,1,0,0,0,0,1,0,1,0,0,1,0,0,1);
```

and this can also be combined with the transformations between sum- and integral-notations:

```
 CFunction H;
 Local F = H(2,3,3,1,5,2,3,3);
 Transform,H,tointegralnotation(1,last)
            ,replace(1,last)=(0,1,1,0)
            ,encode(1,last):base=2;
 Print;
 .sort

F =
   H(2989750);
```

```
 Transform,H,decode(1,22):base=2
            ,replace(1,last)=(0,1,1,0)
            ,tosumnotation(1,last);
 Print;
 .end

F =
   H(2,3,3,1,5,2,3,3);
```

Similarly one can use the transform statement to determine the dual of a MZV:

```
 CFunction H;
 Local F = H(0,1,0,0,1,0,0,1,1,0,0,0,0,1,0,1,0,0,1,0,0,1);
 Transform,H,replace(1,last)=(0,1,1,0)
            ,reverse(1,last);
 Print;
 .end

F =
   H(0,1,1,0,1,1,0,1,0,1,1,1,1,0,0,1,1,0,1,1,0,1);
```

There are still a few important points left, but let us start looking now at the actual program. The file is mzv.frm and its beginning is

```
#-
#ifndef `WEIGHT'
#define WEIGHT "11"
#endif
#ifndef `ORDER'
#define ORDER "X1"
#endif
#ifndef `GROUPING'
#define GROUPING "{2^((`WEIGHT'-1)/2)}"
#endif
*
#include mzv.h
#message Run with Weight = `WEIGHT', GROUPING = `GROUPING'\
                                          , ORDER = `ORDER',
.global
```

This tells us that the default weight is 11. One may specify another weight when form is

called as in

```
form -d WEIGHT=14 mzv > mzv.log &
```

The order and grouping we will discuss when it becomes relevant. Here we just set default values. Next we include the library file mzv.h. This will contain all relevant declarations and procedures.
The #message instruction puts a message in the file. In this case this message tells what run we have so that you can check in your output files what you have been running.
The .global just makes sure we will not forget what we read from mzv.h. Thusfar this is a typical startup of a FORM program. Next we have to define the master expression:

```
Off Parallel;
L   FF = E(‘WEIGHT’-2,1);
repeat id E(n?pos_,?a) = E(n-1,0,?a)+E(n-1,1,?a);
.sort
On Parallel;
#call duality(E)
.sort
```

The ‘Off Parallel’ and ‘On Parallel’ statements concern TFORM and will be discussed once

we start running big programs.
First we need to generate all objects that are strings of ones and zeroes of which the last is one and the first is zero. This is done in the first module. In the next module we call a procedure that makes sure that only the elements survive of which the dual element would not be lower in the ordering we prefer. This ordering is that we prefer the lowest number of ones (lowest depth), or when the depth is half the weight, we prefer the element that comes lexicographically first.
This duality procedure is not completely trivial because of the requirement of minimal depth. This may involve counting the arguments that are one. The procedure we use is

```
#procedure duality(H)
*
*   Applies duality to finite non-alternating H values in one
*   Uses that weight = number of arguments in integral notation
*              depth  = number of arguments in sum notation
*   Assumes that the input is in integral notation.
*   The input may contain more than a single H.
*
  repeat;
```

```
if ( count(‘H’,1) > 0 );
 id,once,‘H’(?a) = R(?a)*R1(?a)*R2(nargs_(?a));
 Transform,R1,tosumnotation(1,last);
 id R1(?a)*R2(x?) = R2(x-2*nargs_(?a));
 if ( match(R2(x?pos_)) ); * weight > 2*depth
   id R(?a) = R3(?a);
 elseif ( match(R2(0)) );  * weight = 2*depth
   id R(?a) = R(?a)*R1(?a);
   Transform,R,reverse(1,last)
              ,replace(1,last)=(0,1,1,0);
   id R(?a) = R1(?a);
   id R1(?a)*R1(?b) = R3(?a);
 else;        * Must replace. weight < 2*depth
   Transform,R,reverse(1,last)
              ,replace(1,last)=(0,1,1,0);
   id R(?a) = R3(?a);
 endif;
 id R2(x?) = 1;
```

```
    endif;
  endrepeat;
  id R3(?a) = 'H'(?a);
#endprocedure
```

We do the H-functions one by one, using the function R3 to store results. As it is, terms that are not self-dual will now occur twice in our expression. This is solved by dropping the coefficient after the terms have been sorted. That is then done in the next module.

```
DropCoefficient;
id  E(?a)=E(?a)*H(?a);
#call frombasis('WEIGHT')
*
* The following enforces a better ordering inside the equations.
*
#call convert(E,'WEIGHT',0)
#call convert(H,'WEIGHT',0)
*
*   The bracket is essential!
*
```

```
B+   E;
.sort
Off Statistics;
Hide FF;
.sort
```

The next statement splits each term into E, the part outside the brackets and H, the part inside the bracket, which is our initial knowledge. If we know already what the basis will be, we can substitute that now. This is not as farfetched as it may seem, because there are some guesses about the structure of the basis. If we do not have such information, the frombasis routine will do nothing. We know however already that for weight 2 the element `H(0,1)` is better represented by the symbol `z2` and we can do something similar for the odd single sums. The way the complete program is constructed, at the end it suggests a number of missing basis elements if we have not provided a full basis. If we specify too many basis elements, the program will eventually crash. The beginning part of the frombasis procedure looks like

```
#procedure frombasis(WW)
*
#switch ‘WW’
#case 2
```

```
id    H(0,1) = z2;
#break
#case 3
id    H(0,0,1) = z3;
#break
#case 5
id    H(0,0,0,0,1) = z5;
#break
#case 7
id    H(0,0,0,0,0,0,1) = z7;
#break
#case 8
id    H(0,0,0,0,1,0,0,1) = z5z3;
#break
#case 9
id    H(0,0,0,0,0,0,0,0,1) = z9;
#break
#case 10
```

```
id    H(0,0,0,0,0,0,1,0,0,1) = z7z3;
#break
```

For the higher weights there may be more basis elements for each weight. Often the choice is not unique. The choice we made is based on observations and guessed expectations and is explained in the paper. Any proof of what would be a good minimal depth basis would make an excellent paper.

After substituting the basis elements we convert the arguments of the E and H functions to a single argument to make lookup faster. The convert procedure looks like

```
#procedure convert(H,w,par)
#if ( 'par' == 0 )
  Transform,'H',replace(1,last)=(0,1,1,0)
                ,encode(1,last):base=2;
#elseif ( 'par' == 1 )
  Transform,'H',decode(1,'w'):base=2
                ,replace(1,'w')=(0,1,1,0);
#endif
#endprocedure
```

and is basically code we saw before. It should be noted that the weight is necessary in the

decode part to ensure that the proper number of binary digits is obtained.

Then we bracket the expression in terms of the function E and the + in the bracket causes FORM to make an index of where the brackets may be found. This enables speedy lookup. Finally the expression is put in the hide file so that it will not need to be moved around in the next modules.

Next we have to generate the equations. As we saw, the idea is to not have all equations in existence at the same time. On the other hand, it turns out that treating the equations one by one is not a good idea either, because that would mean that we have to manipulate the master expression as many times as there are equations. For the higher weights this is extremely wasteful and time consuming. Hence we will generate the equations in groups. Then we use first a Gaussian elimination above and below the diagonal inside each group, after which we apply the results to the master expression. The optimal number of elements of each group has been investigated and is about the square root of the number of objects we have to compute. This is the default for the value of the preprocessor variable GROUPING.

Before we go back to the main program we will have a look at how we solve a set of equations by Gaussian elimination.

Assume that we have jj equations $F_1 \cdots F_{jj}$. And assume they have been bracketted in terms of the variables we want to solve. In our case this is the function H. We give the variable jj in FORM the name `$jj`. The code would be

```
#$v = 0;
#do k1 = 1,`$jj'
 #if ( termsin(F`k1') != 0 )
  InParallel;
```

```
    Skip F‘k1’;
    #$v = $v + 1;
    #$vna‘$v’ = ‘k1’;
    #$fb‘$v’ = FirstBracket_(F‘k1’);
    #$fv = F‘k1’[$fb‘$v’];
    #$fv = 1/(‘$fv’);
    #$fr‘$v’ = -F‘k1’*(‘$fv’)+‘$fb‘$v’’;
    id ‘$fb‘$v’’ = ‘$fr‘$v’’;
    B   H;
    .sort:Preparing ‘$dcount’;
    #$dcount = $dcount-1;
   #endif
  #enddo
```

As you can see, there is lots of activity with $-variables in the preprocessor. What is happening?
Because equations may become zero during the elimination process, we keep a counter for the number of equations left (`$v`) and we keep a list of their numbers (`$vna‘$v’`). This way we can deal with only nontrivial equations at a later stage.

For a Gaussian elimination with N equations we have to go through the system N times. Hence the #do loop which will treat the equations in order. For each (still) nontrivial equation we eliminate one variable. This is done by asking for the equation whose turn it is what is the object that is outside brackets in the first term (we put that in the variable `$fb‘$v’`). Next we construct the object that this must be equal to, which is the negative of the remainder, divided by the contents of that bracket. This is put inside `$fr‘$v’`.
And now the fun part: using the preprocessor we create the corresponding id statement to make this substitution. Because we do not want to loose the F expression from which we generate this substitution we have skip it in this module. Then we bracket again in H and go on to the next expression. The variable $dcount keeps track of how many variables still need to be eliminated.
After this code we have eliminated $v variables in this set of equations, both above and below the diagonal. Now we have to bring them to a form in which we can use the results in the master equation. Note however that the information in `$fb‘$v’` and `$fr‘$v’` is not up to date. We need to update it with the code

```
#if ( ‘$v’ > 0 )
  InParallel;
  id  H(?a) = Hfill(?a);
```

```
B   Hfill;
.sort:Setting Hfill `$dcount';
#do k1 = 1,`$v'
  #$fb`k1' = FirstBracket_(F`$vna`k1'');
  #$fv = F`$vna`k1''[$fb`k1'];
  #$fv = 1/(`$fv');
  #$fr`k1' = (-F`$vna`k1''*(`$fv')+`$fb`k1'')*replace_(Hfill,H);
#enddo
#do vv = 1,`$v'
  Fill,`$fb`vv'' = `$fr`vv'';
#enddo
```

in which we generate table elements in the table Hfill. In the generation of the rhs of that fill statement we have replaced Hfill again by H.
Why a table?
Imagine we have a group of 512 equations and no equations vanish. If we would work with id-statements, there would be 512 of them. This means that each term would be subject to 512 pattern matchings. That is very much. If we work with tables and this is a sparse table the whole matching takes place internally and by means of a binary search amoung the table

elements. This would mean at most 10 compares and then only for the index of the table. This may take even less time than a single regular pattern matching.
Now a little timeout. What does the InParallel statement do? If we run sequential FORM it is simply ignored. If, however, we run TFORM or ParFORM it tells the program to use a certain method of parallelization.
Normally TFORM will distribute the terms of an expression over the workers and later gather the results. This gives much overhead, specially when the expressions do not contain many terms. The InParallel statement tells FORM that it should distribute the complete expressions over the workers, and only the complete results are placed back under control of the master processor. This gives far less overhead. Of course this works well only when we have lots of small expressions, roughly of equal size.
Next we substitute the contents of the table in the master expression.

```
On Parallel;
On Statistics;
Drop;
Ndrop FF;
Unhide FF;
id  H(?a) = Hfill(?a);
```

```
id  Hfill(?a) = H(?a);
B+   E;
.sort:substitution(‘DEPTH’-‘type’)-{‘$dcount’+1};
Off Statistics;
Hide FF;
```

For this we have to take FF from the hide buffer (UnHide), tell FORM to run back in normal parallel mode, drop all F‘i’ expressions and let the Hfill table do its work. Elements that have not been substituted are written back to the function H. Again we have to bracket in E and make sure that there is a bracket index (B+). After the sort we put FF back in the hide buffer. We write the statistics when we manipulate the FF expression to allow us to check on the progress of the program.
We will now concentrate on the generation of the equations. We have two types of equations: the shuffle equations and the stuffle equations. The stuffle equations have typically fewer terms and are hence simpler to apply. Hence we generate those before the shuffle equations. But more important is that we build the equations in an order in which the equations with a lower depth come first. Hence we have the biggest part of the program inside a do-loop:

```
#do DEPTH = 2,‘WEIGHT’/2
        .
```

```
            .
    #enddo
```

We have to go the half the weight only due to the duality.

We are going to generate an expression named Gen which will contain terms that represent stuffle equations for the given weight and depth. And we will make sure they are all stuffle equations without divergent sums for those parameters.

```
Off Parallel;
L Gen = E(‘DEPTH’)*EE({‘WEIGHT’-‘DEPTH’-1});
repeat id E(x?{>1},?a) = E(x-1,1,?a);
repeat id EE(x?pos_,?a) = EE(x-1,0,?a);
id    EE(?a) = E(?a);
shuffle,E;
id    E(?a,0) = 0;
id    E(1,?a) = 0;
.sort
On Parallel;
Transform,E,tosumnotation(1,last);
```

First we generate a function E with ‘depth’ ones and a function EE with ‘weight’-‘depth’

zeroes. When we shuffle those we obtain all different functions E with weight arguments of which exactly depth arguments are one. Then we throw away the functions with have a last argument that is zero (illegal for sums) or the first argument a one (divergent). Then we convert to sum notation.
At this point the output looks like

```
~~~entering DEPTH = 2
~~~doing stuffles

   Gen=
      +E(2,9)
      +E(3,8)
      +E(4,7)
      +E(5,6)
      +E(6,5)
      +E(7,4)
      +E(8,3)
      +E(9,2)
      +E(10,1)
```

```
      ;
```

The next part of the program concerns the ordering of the equations. In the case of equations at depth 2 this is not so important, but when we go to large values for the weight and the depth this makes the difference between being able to run the program or not being able to.

```
id  E(?a) = E(?a)*fff(?a);
Multiply replace_(fff,ffs);
#do i = 1,‘DEPTH’
id  ffs(x?,?a) = ffs(?a)*fun‘i’(-x);
#enddo
id  ffs(?a) = 1;
Multiply,(
#do i = 1,‘DEPTH’/2
  +EE(‘i’)*funa0(‘i’)
#enddo
      );
repeat id EE(x?pos_,?a)*E(x1?,?b) = EE(x-1,?a,x1)*E(?b);
id  EE(0,?a) = E(?a);
id  E(1,?a) = 0;
```

```
.sort
DropCoefficient;
id  E(x1?,?a)*E(x2?,?b) = E(x1,?a)*E(x2,?b)*funa2(-min_(x1,x2));
```

First we multiply with the symmetric function fff and next we replace fff by ffs. This looks a bit stupid, but it is not. The problem is that we cannot use the ?a wildcards inside (anti)symmetric functions or tensors, but we do want things to be properly symmetrized inside function ffs before we put its arguments inside the functions fun1,... etc. and because we cannot use

```
id fff(?a) = ffs(?a);
```

we use the replace_ function. Because the fun1 etc functions have been declared in the file mzv.h before the function E, their ordering will take precedence over the ordering of the E functions.
Next we take the E function apart into two E functions by putting the first i arguments in one function and the remaining arguments in a second function. And again we eliminate the impossible and the divergent sums. Again, the various splits are marked by a function funa0. The sorting of these fun functions determines in the end the order in which the equations will be processed.

And now comes the processing loop:

```
.sort
Off Parallel;
#$jj = 0;
#do relation = Gen
.sort
InParallel;
Drop Gen;
  #$jj = $jj+1;
  L   F‘$jj’ = ‘relation’;
  #if ( ‘$jj’ >= ‘GROUPING’ )
      #call solvestuffle
      #$jj = 0;
  #endif
#enddo
#call solvestuffle
#$jj = 0;
.sort
```

Each equation is represented by a term in the expression Gen as in

```
Gen=
   +fun1(-3)*fun2(-4)*fun3(-4)*funa0(1)*funa2(-4)*E(4)*E(4,3)
   +fun1(-3)*fun2(-4)*fun3(-4)*funa0(1)*funa2(-3)*E(3)*E(4,4)
   +fun1(-3)*fun2(-4)*fun3(-4)*funa0(1)*funa2(-3)*E(3,4)*E(4)
   +fun1(-3)*fun2(-3)*fun3(-5)*funa0(1)*funa2(-3)*E(3)*E(3,5)
   +fun1(-3)*fun2(-3)*fun3(-5)*funa0(1)*funa2(-3)*E(3)*E(5,3)
   +fun1(-3)*fun2(-3)*fun3(-5)*funa0(1)*funa2(-3)*E(3,3)*E(5)
   +fun1(-2)*fun2(-4)*fun3(-5)*funa0(1)*funa2(-4)*E(4)*E(5,2)
   +fun1(-2)*fun2(-4)*fun3(-5)*funa0(1)*funa2(-4)*E(4,2)*E(5)
   +fun1(-2)*fun2(-4)*fun3(-5)*funa0(1)*funa2(-2)*E(2)*E(4,5)
    ...
```

with two functions E that are the sums to be stuffled. The instruction

```
#do relation = Gen
```

is a loop in which the preprocessor variable relation becomes each time one of the terms of the expression Gen. This way we can generate expressions F1,... till we have a full group and then call the procedure solvestuffle to set up the precise equations, solve them and then substitute the results in the master expression. After the loop has been finished there may still

be untreated expressions because $jj did not make it to the value of the variable GROUPING yet. Hence we have to call solvestuffle once more.
The generation of the shuffle equations is slightly more complicated, because we have to include also the equations that are generated by terms of the form `E(1)*E(....)`. On those equations we can use both the stuffle and the shuffle relations, equate them, after which the divergent terms cancel and we have perfectly normal equations left. Also the ordering scheme for the shuffle equations is a bit more involved, but for the rest relatively similar. More remarks about it are in the paper. For experimentation it used the preprocessor variabe ORDER that we saw at the start of the program. Due to time limitations we skip it here. It is explained in the paper and can be looked up in the code in the file mzv.frm.
At this point there is only a very small amount of unexplained code remaining. First the procedure solvestuffle:

```
#procedure solvestuffle
*
* Procedure for stuffle equations for H in 1.
* Apply stuffles (Z variety) and then let clean solve the equations.
*
  .sort
```

```
    InParallel;
    id,many,fun1?funs(?a) = 1;
    id  E(?a)*E(?b) = HH(?a)*HH(?b) - H(?a)*H(?b);
    Transform,HH,tointegralnotation(1,last);
    Stuffle,H+;
    .sort:Stuffling-`$count';
    InParallel;
    Transform,H,tointegralnotation(1,last);
    #call clean(st)
  #endprocedure
```

After the equations enter in order we do not need the fun functions any longer. Hence we eliminate them. Then we construct the actual equations in which HH are the lower weight functions and H is used for the stuffling. HH has to be in integral notation and after the stuffling we also convert H to integral notation. And after that we have our equations and let clean solve them. The corresponding solveshuffle procedure is rather similar:

```
  #procedure solveshuffle
  *
  * Procedure for stuffle equations for H in 1.
```

```
  * Apply shuffles and then let clean solve the equations.
  *
    .sort
    InParallel;
    id,many,fun1?funs(?a) = 1;
    Transform,E,tointegralnotation(1,last);
    id  E(?a)*E(?b) = HH(?a)*HH(?b) - H(?a)*H(?b);
    Shuffle,H;
    .sort:Shuffling-`$count';
    InParallel;
    #call clean(sh)
  #endprocedure
```

For the shuffles everything has to be in integral notation.
Finally there is the procedure solve. Its startup is

```
  #procedure clean(type)
  *
  *   The handling of the equations once they have been
  *   properly constructed.
```

```
*
  id  HH(?a) = H(?a);
  #call makeHfinite
  .sort: after makeHfinite(‘DEPTH’-‘type’);
  InParallel;
  if ( count(H,1,ln2,1,Sinf,1) > 1 );
    #call duality(H)
    #do k = 1,‘WEIGHT’-1
      id  H(n1?,...,n‘k’?) = mzv‘k’(n1,...,n‘k’);
    #enddo
  else;
    #call duality(H)
    #call frombasis(‘WEIGHT’)
  endif;
  .sort: finite(‘DEPTH’-‘type’);
  InParallel;
*
*  Now a trick to be able to substitute the H-functions of the proper
```

```
*  weight by the contents of the appropriate bracket in FF.
*
  #call convert(H,‘WEIGHT’,0)
  id  H(?a) = R(E(?a));
  id  R(n?$n) = FF[$n];
*
*  Bracket in H so that we may take the first bracket
*
  B   H;
  ModuleOption,local,$n;
  .sort: setup(‘DEPTH’-‘type’)-‘$count’;
  ClearTable Hfill;
  Off Statistics;
  Off Parallel;
```

The first thing to do is to deal with the special equations that have divergent objects. We will skip that procedure by lack of time. It is in the file mzv.h and can handle much more complicated cases than we treat here. Basically it was used to show that the single and simple divergences were all we need.

Next we want to replace the objects of lower weight by their values in the tables of lower weight MZV's that we should construct before we run the program for a given weight. These tables have been declared and read when the file mzv.h was included. This means that if one would like to run the program for weight 20, one should have the tables for the weights 1,...,19 available. And because we have used duality to keep the size of the tables to a minimum, we have to apply duality here as well before we can make the substitution. The tables up to weight 10 are included in the file mzvtables.h. The higher tables have to be constructed by the user or can be picked up from the datamine. They have a tendency to become rather big for the higher weights (like 1.8 Gbytes for weight 21). Hence we do not give all of them in the course site and the ones we give have been treated by bzip2 (gives better results than gzip, even though it is slower).

For the single H occurrences of the weight we are computing we then check first whether we have already a definition of the basis elements. This procedure has been prepared with entries that were obtained after each run and studying the remaining elements, possibly then looking at whether it was better to use other elements to get some pattern in this system. The pattern is that we need all Lyndon words composed of odd integers greater than one that add up to the weight, but some have to be replaced by an element in which we subtract one from the first two indexes and paste two ones at the end. Why this is, is still a great mystery. It seems

to be related to something called 'pushdown' by Broadhurst and is discussed in the paper. Next we encode the arguments of the remaining H functions so that they have only a single argument. This speeds up the calculation enormously. After this we apply the code for the Gaussian elimination that we saw before to the master expression. This involves the use of a $-variable. This variable will be defined for each individual term. In sequential FORM this gives no problems, but in TFORM this would give a random order of updating a central administration. In such a case TFORM would refuse to run the module in parallel. But it so happens we really want to run this module in parallel, and there is no need for putting this variable in a central administration. Hence we let TFORM know that this variable will be used only locally inside each term and its value afterwards is unimportant. This is done with the moduleoption statement. This statement (at least these $ options) are ignored by sequential FORM.

First we have been running the program with (sequential) FORM. The times are on one of the newer (19-feb-2014) Intel processors.

W	Time at end
11	0.60 sec out of 0.61 sec
12	0.89 sec out of 0.91 sec
13	1.67 sec out of 1.67 sec
14	7.61 sec out of 10.65 sec
15	18.09 sec out of 18.11 sec
16	91.43 sec out of 91.52 sec
17	262.41 sec out of 262.64 sec
18	1689.11 sec out of 1690.60 sec
19	4625.14 sec out of 4629.17 sec
20	41265.26 sec out of 41300.92 sec
21	105747.29 sec out of 105839.03 sec

As you see, this runs up steeply. Hence it might be interesting to see what we can do with TFORM, the multithreaded version. We used

```
TFORM -w24 mzv > mzv.log &
```

and the -w24 option indicates that TFORM should use 24 worker threads. This can result in spectacular lines when looking at the activity in top:

```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
14169 t68       20   0  184g  77g 1428 S 2397.7 20.5  16063:01 tform
```

This program was given 184 Gbytes of memory, but it would have run with much less as well. In that case it would have used the disk. As it is, the program is not particularly memory hungry. The weight 18 expression for instance never needs more than 85 Mbytes. The weight 22 expression makes it up to 4.5 Gbytes.

The TFORM statistics are

W	Time at end
11	0.43 sec + 1.25 sec: 1.68 sec out of 0.53 sec
12	1.03 sec + 3.56 sec: 4.60 sec out of 1.28 sec
13	2.09 sec + 7.64 sec: 9.74 sec out of 2.42 sec
14	5.22 sec + 27.07 sec: 32.30 sec out of 6.56 sec
15	12.67 sec + 68.64 sec: 81.31 sec out of 15.13 sec
16	45.02 sec + 309.38 sec: 354.40 sec out of 59.27 sec
17	101.16 sec + 847.60 sec: 948.77 sec out of 131.73 sec
18	332.75 sec + 3925.15 sec: 4257.91 sec out of 523.21 sec
19	716.25 sec + 10314.66 sec: 11030.91 sec out of 1201.15 sec
20	2675.87 sec + 66838.26 sec: 69514.13 sec out of 5759.41 sec
21	6832.57 sec + 169968.61 sec: 176801.19 sec out of 13332.54 sec
22	32290.60 sec + 2111577.46 sec: 2143868.07 sec out of 123851.59 sec

When you compare this, you see that for the lower weights TFORM is actually slower. For such small jobs the overhead is obviously too big. What is also clear is that the total amount of CPU time used in TFORM is much bigger than the CPU time used in sequential FORM.

The precise nature of this is not always understood. We can also see that the time of the master processor is a sizable fraction of the real execution time although this becomes better for the higher weights. This indicates that TFORM can still be improved. Ideal would be that the master processor does not have much to do.

Another way to see the efficiency of the parallelization is to vary the number of workers.

Weight	workers	Time at end
20	24	2628.24 sec + 66821.74 sec: 69449.98 sec out of 5722.42 sec
20	20	2637.25 sec + 62331.76 sec: 64969.01 sec out of 6035.57 sec
20	16	2570.70 sec + 56982.59 sec: 59553.29 sec out of 6304.62 sec
20	12	2496.42 sec + 54214.88 sec: 56711.31 sec out of 7256.94 sec
20	8	2402.49 sec + 50515.10 sec: 52917.60 sec out of 8951.90 sec
20	4	2122.24 sec + 47252.28 sec: 49374.52 sec out of 14054.96 sec
20	2	2156.57 sec + 44627.86 sec: 46784.44 sec out of 24455.02 sec
20	1	11778.75 sec + 36006.46 sec: 47785.21 sec out of 47026.03 sec
20	0	43219.45 sec + 0.00 sec: 43219.45 sec out of 43250.09 sec
20	-	41265.26 sec out of 41300.92 sec

Here a dash indicates sequential FORM, zero workers indicates TFORM but without workers and hence the master is doing all the work. One worker is TFORM with a single worker. This is actually rather stupid, because things would go faster if the master does all, but this option is available for testing purposes.