1 Libraries

When version 1.0 of FORM was designed in general the quality of libraries was very poor. The result was that in the first versions of FORM only the most fundamental library functions were used. All others were made as part of the FORM sources, which was a nice feature during the period that FORM was commercial. In version 3 this strategy was abandoned for some simple reasons:

- The quality of the libraries had improved enormously.
- More standard features were needed when FORM became more powerful.
- Some libraries are much work to make oneself. Hence this takes much time that can be used better.
- If it is a rather popular library, someone else will maintain it.
- The algorithms may be better than the ones that can be easily implemented.
- There are probably a few more.

In this session we will be looking at what libraries have been included and how they were included. This should show how one can add more functionality with external libraries.

Which are these libraries? In order of introduction:

- GMP
- gzip
- pthread
- Kaneko's diagram generator
- MPFR

Four of these libraries are provided with any decent distribution of a C or C++ compiler. The diagram generator has been provided by Toshikaki Kaneko. It was originally used in the Grace system for the automated calculation of reactions, but he has reprogrammed it in C++.

There are also some 'semi-libraries': code that is used as if it is a library, but has been programmed specifically for FORM. Examples are the code for optimizations, the rational polynomial routines and the code for floating point arithmetic. We will treat the floating point routines last.

The GMP and gzip libraries involve only internal coding. They do not have external commands, with exception of specifying the compression level for gzip. Hence we will treat them first. The later libraries involve external features and hence required many more steps during implementation. It is hoped, that after this users can implement more libraries when they need them for their work. Alternatively one might implement more features of these libraries.

1.1 GMP

To understand the first use of the GMP (GNU Multi-Precision) library, one has to understand some of the internal workings of arithmetic operations in FORM and its inherent limitations. It has been known for a long time that the best algorithms for multi-precision calculations depend on the desired precision. For more information about this one should consult volume 2 of "the art of computer programming" by D. Knuth. Originally FORM was prepared for relatively 'small' numbers in the coefficients and hence the simple classical algorithms were used. But during the 1990's the computations of moments of splitting and coefficient functions in DIS made the dealing with much larger numbers necessary. At around the same time the GMP library started to become a standard.

Let us have a look how the native arithmetic works in FORM. Each number is an array of unsigned words, in which a word is half the size of the natural word size in the given compiler/computer. This means that originally a FORM word was 16 bits (for a 32 bits processor) and nowadays, with 64-bit hardware, this size is 32 bits. Why? We want to be able to multiply and divide without needing extensions to the compiler like 'long long' that were originally not standard.

In the current mode we can pack a word into a long and then multiply two longs, after which we can take the resulting long apart easily. Similarly we can set up divisions. In this way we can construct multiplications and divisions of as big a numbers as we like. The arrays are built up with the least significant word in the zero element. We use the classical algorithms for the basic operations. Normally this works quite well. The operation that is most bothersome is the GCD operation. In principle there exist two variations of the Euclidean algorithm. One is the classical one, and one is the binary algorithm. This algorithm works as follows:

```
10011100    100111    1100      11      11     11     0                156
11011000     11011   11011   11011   11000     11                     216
               100                                 11      -> 1100     12
```

Step 1: Take out the trailing zeroes. The minimum number of them determines the powers of 2 in the GCD. Step 2: subtract the two numbers. Take out trailing zeroes in the difference. If the answer is zero, we have the GCD after multiplying with the powers of two in step 1. Step 3: Keep the two smallest numbers. Step 4: Go back to step 2.
It looks very elegant, but once the shifting of the bits through an array of words is taken into account, it depends on the low level instruction set of the computer whether it is faster or slower than the regular algorithm, and even when it is faster, it is not faster by very much.

The best algorithm seems to find its origin in the consideration of the generalised Euclidean algorithm. This algorithm works as follows:

In the generalized Euclidean algoritm we use the property that if g is the GCD of x and y, there exists a pair of integers a and b such that

$$ax + by \;=\; g$$

Assume that y is the smaller number. The normal way would now be to compute

$$z \;=\; x\%y$$

and then replace x by y, and y by z, after which we continue until z is zero, making the y of that moment the GCD. The problem is that the division is a very expensive operation for large numbers. During the whole procedure one can keep track of the ever more accurate values of a and b. In the end one obtains their values as well. This can be very useful when computing for instance $1/x$ when calculus is modulus a prime number.

The trick is now to take in each step only the leading one or two words of each number. This may mean that z is not entirely correct, but that makes no difference in the determination of the new numbers, as they still have the same GCD:

$$\begin{aligned} x' &\leftarrow max(x - zy, y) \\ y' &\leftarrow min(x - zy, y) \end{aligned}$$

Because we use half the natural wordsize of the computer, all divisions are done with natural words. This is called the Lehmer-Euclidean algorithm. It sped up the calculations of complicated moments of structure functions by a factor 7 when implemented in FORM in 2006. [1]

[1] D.H.Lehmer: "Euclid's algorithm for large numbers". American Mathematical Monthly, 45:227-233, 1938.

There are two disadvantages to the above setup of having FORM do everything 'by itself'. The first is that for very large numbers there are better algorithms. These would typically be numbers that take more than a few hundred words. The second is that some operations are better done in assembler language. It is one of the policies wrt FORM to only use higher language in coding and to avoid machine dependencies. If one uses assembler language, one can work with whole computer words, because after a multiplication the leading part, that would be lost in a higher language, can be found in another register, and similarly one can do divisions. This cuts the time of many operations by a factor 2-4. Both these problems are taken into account in a library like the GMP library. Hence, at a given moment during working with version 2 of FORM it was decided to use the GMP library when the precision of the numbers becomes more than a few FORM words. The reason of this turnover point is that to use the GMP library, a conversion is needed, because GMP uses a different notation. It is not clear whether the choice of turnover point is optimal. It may be noted that currently a good C library offers facilities that address the problem of assembler language having an advantage wrt. those registers.

All code for this implementation of the GMP library is contained in the file reken.c and form3.h. It is controled with the macro WITHGMP. If this macro is not defined FORM uses its own algorithms/code. This is of course historical, because there was a period that on some computers the GMP library might not be available.

Example (from the routine that multiplies lengthy numbers):

```
#ifdef WITHGMP
    if (na > 3 && nb > 3) {
/*      mp_limb_t res;  */
        UWORD *to, *from;
        int j;
        GETIDENTITY
        UWORD *DLscrat9 = NumberMalloc("MulLong"),
              *DLscratA = NumberMalloc("MulLong"),
              *DLscratB = NumberMalloc("MulLong");
#if ( GMPSPREAD != 1 )
        if ( na & 1 ) {
            from = a; a = to = DLscrat9; j = na; NCOPY(to, from, j);
            a[na++] = 0;
            ++*nc;
        } else
#endif
        if ( (LONG)a & (sizeof(mp_limb_t)-1) ) {
            from = a; a = to = DLscrat9; j = na; NCOPY(to, from, j);
        }
```

```
#if ( GMPSPREAD != 1 )
        if ( nb & 1 ) {
            from = b; b = to = DLscratA; j = nb; NCOPY(to, from, j);
            b[nb++] = 0;
            ++*nc;
        } else
#endif
        if ( (LONG)b & (sizeof(mp_limb_t)-1) ) {
            from = b; b = to = DLscratA; j = nb; NCOPY(to, from, j);
        }

        if ( ( *nc > (WORD)i ) || ( (LONG)c & (LONG)(sizeof(mp_limb_t)-1) )
) {
            ic = DLscratB;
        }
        if ( na < nb ) {
            /* res = */
            mpn_mul((mp_ptr)ic, (mp_srcptr)b, nb/GMPSPREAD,
                    (mp_srcptr)a, na/GMPSPREAD);
        } else {
```

```
            /* res = */
            mpn_mul((mp_ptr)ic, (mp_srcptr)a, na/GMPSPREAD,
                    (mp_srcptr)b, nb/GMPSPREAD);
        }
        while ( ic[i-1] == 0 ) i--;
        *nc = i;
/*
        if ( res == 0 ) *nc -= GMPSPREAD;
        else if ( res <= WORDMASK ) --*nc;
*/
        if ( ic != c ) {
            j = *nc; NCOPY(c, ic, j);
        }
        if ( sgn < 0 ) *nc = -(*nc);
        NumberFree(DLscrat9,"MulLong"); NumberFree(DLscratA,"MulLong");
        NumberFree(DLscratB,"MulLong");
        return(0);
    }
#endif
```

Here GMPSPREAD is the number of FORM words that fit inside a GMP word.

We will see more use of the GMP library in the part about floating point numbers and their arithmetic.

1.2 gzip

Another problem in the calculations of Mellin moments of structure functions was the size of the expressions. This was mostly the case during sorting, when the size of the sort file can be several times the size of the input or output. Of course FORM uses already some compression in its expressions, but by studying these sort files it became clear that gzip could compress them further by typically a factor 4 to 5. This was considerable in the days that disks had rather limited sizes. In addition, with old style disks, the reading operation is relatively slow, because for a random read there may be a waiting time of about 8 msec. With the SSD disks this delay has mostly vanished. Of course gzip is not a very simple program, and it may well be that in the future it could obtain even better algorithms. This led to the decision to use the gzip library, rather than trying to program its algorithms natively into FORM.

There are several problems connected to the implementation of the gzip compression inside FORM. gzip is meant to compress complete files, and does not allow access to random points inside the compressed file. This means that it cannot be applied to the scratch files, because when we want to access the contents of a bracket, we may have a bracket index to speed this up and that would imply reading terms from the given location. This leaves the sort files, from which the patches are read sequentially only, and potentially the saved files, provided they are unpacked when copied to the .str file. This last feature has not been implemented as of yet, but should not be very difficult. It would need modifications only in the file store.c and maybe some extra variable in one of the data structs.

Of course one could use gzip on the .sav files externally, but that means that one would have to unzip the whole

file before using it, even if one needs only a single expression from it.

The biggest benefit is to be found in the sort file. Here we should be aware that each patch in each sort file (when using TFORM) should be an independent stream. Hence if we are merging 32 patches from each of the sort files in a TFORM program with 16 workers, we have 512 gunzippings going on at the same time. Each of these needs buffers. Fortunately the gzip library has been set up properly for this and we only have to worry about the bookkeeping in the FORM side. The file compress.c has the interface with the gzip library, and in sort.c we have to decide whether to call the proper routines. There is a macro GZIPDEBUG that can activate code to aid in the debugging by printing information about the value of critical variables.

It should be noted, that there exists a better compression that does preserve the possibility to enter into an expression at intermediate points. This one works at the byte level and utilises the fact that there are very many zero bytes inside the FORM expressions. And also this compression can still be combined, both with the original FORM compression at the term level, and the gzip compression afterwards. It will be quite some work though to implement it. It would however compete with the most economical ways to store multivariate polynomials, while at the same time be applicable for all internal objects. It also would allow the use of a bracketindex (of course without the gzip step). I never had the time to implement it though.

1.3 pthread

When parallelisation became more of an issue, at first the group in Karlsruhe decided to make a version of FORM which could run operations in parallel. This became ParFORM. It would run on the special computers that the group had, and sped up calculations enormously. The main issue with ParFORM was the amount of data that had to be transferred between the various computers. This needed relatively expensive hardware. At a later point the computer industry came up with chips with multiple processors and shared memory. This solves the worst part of the data transfer, because now one may only have to pass a pointer, but it required a different model for the parallelisation, based on threads. For this TFORM was constructed. This uses the multiple thread conventions of POSIX threats, and because I had picked up a limited experience with parallelisation in the early 90's at Fermilab, FORM was more or less ready for this, even though the sources did have to be cleaned up a lot. Such cleanups always run the risk of introducing errors, and hence it is not excluded that some still exist, after all these years.

The first cleanup meant that the global variables had to be strictly divided over a number of substructs, in such a way that local variables for each threat would be in one set of substructs, while all other substructs would be in another set. Once this was done, for sequential FORM all substructs could be part of the main global struct A, while for TFORM, the private substructs could be in a separate struct for each of the threats while the others would remain inside A. The next step, which is essential here, was to create a few macro's in such a way that both for FORM and for TFORM the sources remain identical. These have been explained before. The interface to the POSIX library is inside the file threads.c:

```
## Variables :
## Identity :
## StartHandleLock :
## StartAllThreads :
## InitializeOneThread :
## FinalizeOneThread :
## ClearAllThreads :
## TerminateAllThreads :
## MakeThreadBuckets :
## GetTimerInfo :
## WriteTimerInfo :
## GetWorkerTimes :
## UpdateOneThread :
```

```
## LoadOneThread :
## BalanceRunThread :
## SetWorkerFiles :
## RunThread :
## RunSortBot :
## IAmAvailable :
## GetAvailableThread :
## ConditionalGetAvailableThread :
## GetThread :
## ThreadWait :
## SortBotWait :
## ThreadClaimedBlock :
## MasterWait :
## MasterWaitThread :
## MasterWaitAll :
## MasterWaitAllSortBots :
## MasterWaitAllBlocks :
## WakeupThread :
## WakeupMasterFromThread :
## SendOneBucket :
## InParallelProcessor :
```

```
## ThreadsProcessor :
## LoadReadjusted :
## SortStrategy :
## PutToMaster :
## SortBotOut :
## MasterMerge :
## SortBotMasterMerge :
## SortBotMerge :
## IniSortBlocks :
## DefineSortBotTree :
## GetTerm2 :
## TreatIndexEntry :
## SetHideFiles :
## IniFbufs :
## SetMods :
## UnSetMods :
## find_Horner_MCTS_expand_tree_threaded :
## optimize_expression_given_Horner_threaded :
```

Many routines have of course a name that explains what they do. An interesting routine is RunThread. This runs the 'show' for each thread.

```
    B = InitializeOneThread(identity);
    while ( ( wakeupsignal = ThreadWait(identity) ) > 0 ) {
        switch ( wakeupsignal ) {
/*
            ## STARTNEWEXPRESSION :
            ## LOWESTLEVELGENERATION :
            ## FINISHEXPRESSION :
            ## CLEANUPEXPRESSION :
            ## HIGHERLEVELGENERATION :
            ## STARTNEWMODULE :
            ## TERMINATETHREAD :
            ## DOONEEXPRESSION :
            ## DOBRACKETS :
            ## CLEARCLOCK :
            ## MCTSEXPANDTREE :
            ## OPTIMIZEEXPRESSION :
*/
            default:
```

```
                MLOCK(ErrorMessageLock);
                MesPrint("Illegal wakeup signal %d for thread %d",
                                        wakeupsignal,identity);
                MUNLOCK(ErrorMessageLock);
                Terminate(-1);
                break;
        }
        /* we need the following update in case we are using checkpoints. then we
           need to readjust the clocks when recovering using this information */
        timerinfo[identity] = TimeCPU(1);
    }
EndOfThread:;
```

The worker is in a loop in which it receives a signal, and takes action depending on what the signal is. Of course signals are relatively expensive, and hence one wants to minimize their use. An example

```
            case CLEANUPEXPRESSION:
/*
                Cleanup everything and wait for the next expression
*/
                if ( AR.outfile->handle >= 0 ) {
                    CloseFile(AR.outfile->handle);
                    AR.outfile->handle = -1;
                    remove(AR.outfile->name);
                    AR.outfile->POfill = AR.outfile->POfull = AR.outfile->PObuffer;
                    PUTZERO(AR.outfile->POposition);
                    PUTZERO(AR.outfile->filesize);
                }
                else {
                    AR.outfile->POfill = AR.outfile->POfull = AR.outfile->PObuffer;
                    PUTZERO(AR.outfile->POposition);
                    PUTZERO(AR.outfile->filesize);
                }
                {
                    CBUF *C = cbuf+AT.ebufnum;
                    WORD **w, ii;
                    if ( C->numrhs > 0 || C->numlhs > 0 ) {
```

```
            if ( C->rhs ) {
                w = C->rhs; ii = C->numrhs;
                do { *w++ = 0; } while ( --ii > 0 );
            }
            if ( C->lhs ) {
                w = C->lhs; ii = C->numlhs;
                do { *w++ = 0; } while ( --ii > 0 );
            }
            C->numlhs = C->numrhs = 0;
            ClearTree(AT.ebufnum);
            C->Pointer = C->Buffer;
        }
    }
    break;
```

It should be noted here that AR and AT are two of the three substructs that are private to the worker. Details about some of the variables can be found in the presentations of last year (2023), but we give the layout of the main structs anyway:

```
#ifdef WITHPTHREADS
typedef struct AllGlobals {
    struct M_const M;
    struct C_const Cc;
    struct S_const S;
    struct O_const O;
    struct P_const P;
    struct X_const X;
    PADPOSITION(0,0,0,0,sizeof(struct P_const)+sizeof(struct X_const));
} ALLGLOBALS;

typedef struct AllPrivates {
    struct R_const R;
    struct N_const N;
    struct T_const T;
    PADPOSITION(0,0,0,0,sizeof(struct T_const));
} ALLPRIVATES;

#else

typedef struct AllGlobals {
```

```
    struct M_const M;
    struct C_const Cc;
    struct S_const S;
    struct R_const R;
    struct N_const N;
    struct O_const O;
    struct P_const P;
    struct T_const T;
    struct X_const X;
    PADPOSITION(0,0,0,0,sizeof(struct P_const)+sizeof(struct
T_const)+sizeof(str
} ALLGLOBALS;

#endif
```

and then the macro's

```
   #ifdef WITHPTHREADS
   #define AC A.Cc
   #define AM A.M
   #define AO A.O
   #define AP A.P
```

```
#define AS A.S
#define AX A.X
#define AN B->N
#define AR B->R
#define AT B->T
#define AN0 B0->N
#define AR0 B0->R
#define AT0 B0->T
#else
#define AC A.Cc
#define AM A.M
#define AN A.N
#define AO A.O
#define AP A.P
#define AR A.R
#define AS A.S
#define AT A.T
#define AX A.X
#endif
```

The main problem outside the threads.c file is to pass the private struct to the routines that are run by the workers if they need any information inside this struct. The regular way is to pass its address, if the calling routine has this address available. If not there is the macro GETIDENTITY, but that one has to make a call to a POSIX library routine, which is more expensive than just passing an address. But because sequential FORM does not use such a private struct we need again a flexible definition of macro's to keep the sources identical:

```
#ifdef WITHPTHREADS
#define PHEAD  ALLPRIVATES *B,
#define PHEAD0 ALLPRIVATES *B
#define BHEAD  B,
#define BHEAD0 B
#else
#define PHEAD
#define PHEAD0 VOID
#define BHEAD
#define BHEAD0
#endif
```

after which we declare for instance the function Generator

```
extern WORD   Generator(PHEAD WORD *,WORD);
```

and its header

```
WORD Generator(PHEAD WORD *term, WORD level)
```

and we use it with

```
if ( *termout && Generator(BHEAD termout,level) < 0 ) goto GenCall;
```

while GETIDENTITY is given by

```
#define GETIDENTITY int identity = WhoAmI(); ALLPRIVATES *B = AB[identity];

ALLGLOBALS A;
#ifdef WITHPTHREADS
  ALLPRIVATES **AB;
#endif
```

With the above macro's nearly all problems wrt keeping one set of sources have been solved. There are of course problems with access to some variables, like dollar variables, specially if they have to be changed at runtime. In that case we need locks. The same holds for writing messages. In code like

```
Print "Problems in term %t";
```

it could happen that several workers will try to write at the same time. This needs locks in order to avoid chaos. And it is even better if one uses:

```
Print "Worker %w: Problems in term %t";
```

Such problems will always occur when one runs in parallel, but output of the workers goes to a single file/memory location. The most complicated lock system is when a multi-worker FORM program crashes, and FORM has to do a multi-worker cleanup in which the signals of the OS have to be held until after the cleanup. It maybe that there are still some race conditions there.

1.4 Simplification

The simplification routines are original FORM code, but this code can be treated almost like an external library. The code was originally made by Jan Kuipers, after which Ben Ruijl has made a number of improvements as part of his thesis work. But because this code is now about 10 years old, it should be possible to improve upon it with AI techniques that were not available in the past.

The algorithms used inside the library have been explained in the literature. Here we will concentrate on how it interfaces with the FORM code. Of course we need a struct that contains all relevant data connected to settings etc.

```
typedef struct {
    union { /* we do this to allow padding */
        float fval;
        int ival[2]; /* This should be enough */
    } mctsconstant;
    int   horner;
    int   hornerdirection;
    int   method;
    int   mctstimelimit;
    int   mctsnumexpand;
    int   mctsnumkeep;
    int   mctsnumrepeat;
```

```
    int   greedytimelimit;
    int   greedyminnum;
    int   greedymaxperc;
    int   printstats;
    int   debugflags;
    int   schemeflags;
    int   mctsdecaymode;
    int   saIter; /* Simulated annealing updates */
    union {
        float fval;
        int ival[2];
    } saMaxT; /* Maximum temperature of SA */
    union {
        float fval;
        int ival[2];
    } saMinT; /* Minimum temperature of SA */
    int    spare;
} OPTIMIZE;

typedef struct {
    WORD  *code;
```

```
    UBYTE *nameofexpr;  /* It is easier to remember an expression by name */
    LONG  codesize;     /* We need this for the checkpoints */
    WORD  exprnr;       /* Problem here is: we renumber them in execute.c */
    WORD  minvar;
    WORD  maxvar;
    PADPOSITION(2,1,0,3,0);
} OPTIMIZERESULT;
```

These structs will be placed inside the A.O substruct (O is for output) because the optimization is a way to present the output.

The options are of course read in the Format statement which is dealt with by the compiler. The statement is then properly read in the file compcomm.c. Of course the default values are set in the startup.c file. The actual optimization is done by a preprocessor instruction.

Because the actual optimization is done over expressions that contain only symbols, we use the ToSymbol feature of FORM. In addition we have to find a way to print the output with many generated variables. The easiest is to use the feature for extra symbols in an array notation.

1.5 Polynomials

During the 90's, when many of the challenges in using FORM were with moments of structure functions, the PolyFun was invented to make the programs much faster, and the expressions shorter. At a later stage it became important to have rational polynomials, but because those are far harder to implement, it took its time. Personally I had already a version with univariate rational polynomials, but simultaneously Misha Tentyukov was working on a different solution, but using an external program that could be communicated with. This worked for the problems that he had with calculations people were doing in Karlsruhe. Unfortunately this is of course not ideal, because such communications cost a lot of overhead. But for a few years people were using this system. Then Jan Kuipers managed to make a working system for FORM, which at a later stage was improved by Ben Ruijl. Of course, in a multivariate polynomial system the two tricky operations are the GCD and factorisation. Having the GCD operation the PolyRatFun became possible in FORM, but because that GCD operation is rather costly, we had to implement in a way that is should be used as few times as possible. This merits a special dirty flag in the flag field of the PolyRatFun and very careful manipulations when terms are put together and normalised. Another problem with the PolyRatFun can be that its arguments can become rather lengthy when the polynomials become complicated, and one has to manipulate the MaxTermSize very carefully.

What is needed to hook up such a library?

We need of course the regular routines for addition, multiplication etc in the library. But the messy part is inserting the code that calls the manipulation of the coefficients. In particular the change in the size of the arguments of the PolyRatFun causes problems. In the past we could be sure that the sum of two twerms would never take more space that the sum of the spaces of the two terms. This allowed a modest size for the SmallExtension during sorting. Of this we cannot be sure now, and this shows itself during running with complicated multivariate rational coefficients. Often we have to set much bigger sizes for the SmallExtension. Also the value of MaxTermSize may have to be increased. But the last variable cannot be increased arbitrarily, because the sizes of many buffers depend on it. One also has to become careful with the sizes of subfilepatches ,sublargepatches ,sublargesize ,subsmallextension ,subsmallsize ,subsortiosize ,subtermsinsmall in the setup. This is explained in the lecture about the IBP reduction of a generic massless T1 topology in the FORM course.

1.6 Diagram generation

Many projects in the language of FORM involve the generation of Feynman diagrams in one way or another. There are several diagram generators on the market, but sometimes it turns out that an extra effort is needed to get them to do exactly what one needs. The most widely used generator seems to be QGRAF by Paolo Nogueira, but it has been programmed in such a way that it is nearly impossible to customize it. Hence, when it became clear that Toshiaki Kaneko was reprogramming the GRACE diagram generator into C++, it looked very attractive to attach this library to FORM. To this end a collaboration with Kaneko was set up, but it was slowed down a lot by the Covid pandemic. Based on experience with QGRAF, some features have been included making some steps easier. Kaneko also wrote a very clear manual. Hence much of what is needed is now available as if it is native FORM code. Of course such an implementation requires:

- external commands to control parameters,
- special functions,
- new types of variables, like particles,
- a good interface between the library and FORM,
- the possibility to add new features in a relatively painless way.

The question is now how to make FORM talk to the Kaneko library. Such a library needs to know about particles and vertices. Hence the decision was made to introduce such new data types. In addition a very useful feature would be to first run a program just on the topologies of the diagrams, to determine notations for such topologies, and only afterwards run the program with the full set of diagrams. For that the numbering of the topologies should be identical in both cases. The Kaneko generator can do this. When working with QGRAF it always took much more programming and running time to determine the proper topology of the diagrams.

Another feature would be to be able to use more than one model in a single program. Of course one can prepare #include files in which the particles and vertices are defined, but it becomes even easier when there is a variable type called model which a name and its properties, like a list of its particles and its vertices.

The above indicates that we have to start with some compiler work to be able to read these new types of variables. Once they have been entered in the lists of variables, we need commands or functions that do something with them, and hence transfer the proper instructions to the Kaneko library. Hence (in structs.h)

```
typedef struct PaRtIcLe {
    WORD number; /* Number of the function */
    WORD spin;   /* +/- dimension of SU(2) representation */
    WORD mass;   /* Number of symbol or 0 */
    WORD type;   /* -1: anti particle, 1: particle, 0: own antiparticle */
} PARTICLE;

typedef struct VeRtEx {
    PARTICLE particles[MAXPARTICLES];
    WORD couplings[2*MAXCOUPLINGS];
    WORD nparticles;
    WORD ncouplings;
    WORD type;
    WORD error;
    WORD externonly;
    WORD spare;
} VERTEX;
```

```
typedef struct MoDeL {
    VERTEX **vertices;
    WORD *couplings;
    UBYTE *name;
    void *grccmodel;
    WORD legcouple[MAXLEGS+2];
    WORD nparticles;
    WORD nvertices;
    WORD invertices;
    WORD sizevertices;
    WORD sizecouplings;
    WORD ncouplings;
    WORD error;
    WORD dummy;
} MODEL;
```

The particles are actually a special type of vertex. This means that we do not have to introduce too much code for them. We mainly need code for entering their properties. Vertices and models need however a 'flexible' amount of space, and hence involve memory allocations.

First we need routines for reading the Model, Partcle and Vertex declarations. This is in principle straightforward. these routines can be found in the file model.c. To keep things 'simple' a particle is stored as a rather simple vertex. And to be able to recognise particles from vertices, all declarations of particles and vertices should be inside the definition of a model with the particles before the vertices. During the running, when diagrams have been generated the vertices will be indicated by a function node_. In each vertex we have to be able to attach coupling constants, which will be symbols, and there should be provisions to have more than a single one.

The next step is to provide a way to invoke the diagram generator. This is done with the special function diagrams_. In an earlier version we had a more primitive system in which the function topologies_ could determine the topologies for a given reaction. The functions that would be involved are in the file diagrams.c. This approach was however abandoned when it became clear that the Kaneko generator provided nearly everything we needed. Hence the new function diagrams_ was created, but for backward compatibility the topologies_ function still exists. It may be removed in the future if it becomes sufficiently clear that is is no longer being used.

The diagrams_ function is recognised and has a legal set of arguments (this is done in the routine TestSub in generator.c) it is properly flagged and the routine Generator will then call the routine GenDiagrams in the file diawrap.cc. It is a generic way of dealing with libraries written in C++, to have a wrapper file that is also written in a mixture of C and very simple C++. In this file all the FORM interfacing with the grcc.cc file takes place. The only thing we have done to modify the grcc.cc file is to add folding information to make it easier to have a look at its code when needed.

To pass the proper parameters it seemed best to define a number of preprocessor variables with suggestive names, allowing any combination to be passed as a sum of the various options:

```
    PutPreVar((UBYTE *)"ONEPI_",(UBYTE *)("1"),0,0);
    PutPreVar((UBYTE *)"WITHOUTINSERTIONS_",(UBYTE *)("2"),0,0);
    PutPreVar((UBYTE *)"NOTADPOLES_",(UBYTE *)("4"),0,0);
    PutPreVar((UBYTE *)"SYMMETRIZE_",(UBYTE *)("8"),0,0);
    PutPreVar((UBYTE *)"TOPOLOGIESONLY_",(UBYTE *)("16"),0,0);
    PutPreVar((UBYTE *)"NONODES_",(UBYTE *)("32"),0,0);
    PutPreVar((UBYTE *)"WITHEDGES_",(UBYTE *)("64"),0,0);
  /*      Note that CHECKEXTERN is 128 */
    PutPreVar((UBYTE *)"WITHBLOCKS_",(UBYTE *)("256"),0,0);
    PutPreVar((UBYTE *)"WITHONEPISETS_",(UBYTE *)("512"),0,0);
    PutPreVar((UBYTE *)"NOSNAILS_",(UBYTE *)("1024"),0,0);
    PutPreVar((UBYTE *)"NOEXTSELF_",(UBYTE *)("2048"),0,0);
```

The CHECKEXTERN is for debugging purposes.

The file diawrap.cc has a few routines, some of which are called by the regular FORM code, and some are called by the library, like ProcessDiagram.

```
//  ## Includes : diawrap.cc
//  ## LoadModel :
//  ## ConvertParticle :
//  ## ReConvertParticle :
//  ## numParticle :
//  ## ProcessDiagram :
//  ## fendMG :
//  ## ProcessTopology :
//  ## GenDiagrams :
//  ## processVertex :
//  ## GenTopologies :
```

The specific code in these routines is of course not to be presented here. There are more tham 1000 lines.

One feature is still missing: to bring diagrams to a canonical form. There exist libraries for this, but the development has not reached their implementation yet. This canonicalisation is important when diagrams are manipulated during further processing, and lines are modified, or become missing. To determine the new topology can be a time consuming operation, and hence a general and fast library would be appreciated.

1.7 Floating point

The implementation of arbitrary precision floating point numbers was long overdue. This had been planned already in the earliest stages of the development of FORM, but somehow there was never the immediate need, until a few years ago. As explained before, the notation of the coefficients in the terms left room for floating point numbers provided the number of FORM words occupied by them would be even. The odd lengths are for fractions. Unfortunately since then so much code was added, that the prospect of having to add the even case everywhere (potentially more than 1000 places) was not only discouraging, but also the chance that this could be done without introducing serious bugs made me look for a different solution. This was found in the introduction of a new function float_, internally indicated by FLOATFUN. This has the advantage that we only deal with floating point numbers if they are really present, and we do not have to worry about them in cases that they do not occur. There is of course one side condition, and that is that when we do have a floating point number in a term, the rational coefficient needs to be absorbed into it. This means that if a normalized term contains the function float_, the rational coefficient must be either 1/1 or -1/1. We try to absorb the sign of the term in the rational coefficient, because that is much easier for the print routines.

Let us see what we need.

- A statement to activate the floating point numbers and the precision to be used.
- A statement to deactivate the floating point numbers.
- A notation for the arguments of the float_ function. If the arguments do not fulfil this notation, the floating point number is not recognized as such.
- Conversion routines from rational to float et vise versa.
- Routines to read and write floating point numbers.
- Conversion to and from GMP floating point numbers.
- Code in the Normalize routine to combine two or more float_ functions or a float_ and a rational coefficient.
- Code in the sort routines that allows the addition of two terms.
- A number of built-in numerical functions to arbitrary precision. This is of course an open ended project and not necessarily trivial, because the fixed precision functions can use methods that are not applicable for arbitrary precision, like Chebychev expansions.

Because programming arbitrary precision is a lot of work, and may need maintenance when better algorithms are discovered, we use the floating point facilities of the GMP library and when we need to evaluate the standard functions we use also the MPFR library. Both libraries are well documented and part of many standard distributions. They provide us with the basic lower level functions, but we need to hook them up with the complete FORM sources which is quite a lot of work. In addition the MPFR provides us with the 'standard' functions like sin_, cos_ etc.

The first command we need is

```
#StartFloat
#StartFloat Precision
#StartFloat Precision MaxWeight
```

in which the parameters are numbers. Precision indicates the precision of the floating point numbers in bits. MaxWeight indicates the maximum weight of potential Multiple Zeta Values or Euler sums. If there are no parameters the floating point precision is set to DEFAULTPRECISION if it is the first call to StartFloat, and otherwise to the value used in the most recent call to StartFloat. If MaxWeight is not present, its value is set to zero.
It should be noted that currently the floating point system will only work on 64-bits computers.

Because we will also provide routines for Multiple Zeta values (more about that below) we might need to allocate space for arrays that they need during evaluation. The allocations for the MZV's have to be done for each worker separately. Hence they are done in a separate routine SetupMZVTables. The exact allocations seem a bit strange. They will be discussed in the routines for the computation of these MZV's.

The preprocessor instruction

```
#EndFloat
```

closes the floating point system by releasing all allocated buffers for it. This uses the routines ClearfFloat and ClearMZVTables. If a #StartFloat is invoked without ending a previous one, internally FORM calls those two routines first, before starting a new system.

Now let us have a look at how floating point numbers are stored inside the GMP library.

```
typedef struct
{
  int _mp_prec;     Max precision, in number of `mp_limb_t's.
                    Set by mpf_init and modified by
                    mpf_set_prec.  The area pointed to by the
                    _mp_d field contains `prec' + 1 limbs.
  int _mp_size;     abs(_mp_size) is the number of limbs the
                    last field points to.  If _mp_size is
                    negative this is a negative number.
  mp_exp_t _mp_exp; Exponent, in the base of `mp_limb_t'.
  mp_limb_t *_mp_d; Pointer to the limbs.
} __mpf_struct;
```

and with this goes

```
typedef __mpf_struct mpf_t[1];
```

In the current implementation we have

```
sizeof(int) = 4
sizeof(mpf_t) = 24
sizeof(mp_limb_t) = 8,
sizeof(mp_exp_t) = 8,
sizeof a pointer is 8.
```

Somehow we have to get this information into the float_ function and back. This we do with the 4 arguments

```
-SNUMBER _mp_prec
-SNUMBER _mp_size
exponent which can be -SNUMBER or a regular numerical term
the limbs as n/1 in regular term format, or just an -SNUMBER.
```

```
FORM 5.0Beta (Apr  9 2023)                          Run: Thu May 16 10:44:52
2024
    #StartFloat 200
    Symbol x,y;
    Format 60;
    Local F = x*y/3;
    ToFloat;
    Print "  %r";
    .end
  39  1  6  20  1  21  1  116  29  0  -16  4  -16  4  -16  0  20
  0  18  1431655765  1431655765  1431655765  1431655765  1431655765
  1431655765  1431655765  1431655765  1  0  0  0  0  0  0  0
  17  1  1  3

Time =       0.00 sec    Generated terms =          1
               F         Terms in output =          1
                         Bytes used      =        160
  0.00 sec out of 0.00 sec
```

At the same time we have to give the float_ function a protected status with respect to operations that work on nearly all other functions, like Transform, or things like

```
id  f?(x1?,x2?,?a) = f(x1+x2,?a);
```

Also, for not upsetting too many of the internal routines, we place the float_ function as last subterm, before the rational coefficient. This requires of course some extra code in the Normalize routine and the write routines.

Another consideration should be what to do with a float_ function when we use #endfloat. At that moment the function becomes a regular function with no special status. And after a new #StartFloat we have to test whether a float_ function does indeed qualify as a proper floating point function.

At this point we are ready to set up the internal low level routines. There are actually quite a few:

```
#[ Low Level :

    In the low level routines we interact directly with the content
    of the GMP structs. This can be done safely, because their
    layout is documented. We pay particular attention to the init
    and clear routines, because they involve malloc/free calls.

    ## Explanations :
    ## Form_mpf_init :
    ## Form_mpf_clear :
    ## Form_mpf_empty :
    ## Form_mpf_set_prec_raw :
    ## PackFloat :
    ## UnpackFloat :
    ## TestFloat :
    ## FormtoZ :
    ## ZtoForm :
    ## FloatToInteger :
    ## IntegerToFloat :
```

```
    ## RatToFloat :
    ## FloatFunToRat :
    ## FloatToRat :
    ## ZeroTable :
    ## ReadFloat :
    ## CheckFloat :
#] Low Level :
```

Most functions have a name that is rather evident. Z is the GMP type for long integers. These are packed differently from the way FORM packs them and hence we need conversions.

The interesting routine here is FloatToRat. This conversion is done by means of repeated fractions in which the routine keeps track of the remaining accuracy. The fraction terminates if one runs into a number N in 1/N which is bigger than half the remaining accuracy.

```
 #startfloat 200
 Format 72;
 Off Statistics;
 L   F = 355/113+1/10000000000000000000000000000000000000000000000;
 ToFloat;
 Print;
 .sort

F =
   3.14159292035398230088495575221238938053097345132843362831850e+00;

 Hide;
 L   F1 = 1/(F-3);
 Print;
 .sort

F1 =
```

```
   7.062499999999999999999999999999999999999950121093753e+00;

 L   F1 = 1/(F1-7);
 Print;
 .sort

F1 =
   1.600000000000000000000000000000000000012768999999222e+01;

 L   F1 = 1/(F1-16);
 Print;
 .sort

F1 =
   7.831466834214822254373300035704757987174055802239280319081 4e+43;

 Drop F1;
 L   F2 = 3+1/(7+1/16);
 Print;
 .end
```

```
F2 =
   355/113;
```

For cases in which there is indeed a valid fraction this is usually sufficient (provided that the floating point precision suffices as well). When one is not sure there is sufficient accuracy it is best to run with two different floating point accuracies and see whether the fractions change. Usually one can see whether the fractions have the proper structure. If all other fractions have two or three digits in the numerator and the denominator, and one of them has a few hundred digits.....

The next level of routines can be called from the other parts of FORM. They are

```
#[ Float Routines :
    ## SetFloatPrecision :
    ## PrintFloat :
    ## AddFloats :
    ## MulFloats :
    ## DivFloats :
    ## AddRatToFloat :
    ## MulRatToFloat :
    ## SetupMZVTables :
    ## SetupMPFTables :
    ## ClearMZVTables :
    ## CoToFloat :
    ## CoToRat :
    ## ToFloat :
    ## ToRat :
#] Float Routines :
```

The Co routines are the compiler routines for the ToFloat and ToRat statements, while the last two routines execute these statements during runtime.

Sorting introduces extra complications. The fact that we have activated floating point numbers does not imply that all terms have a floating point coefficient. That would only be the case after a ToFloat statement. Hence we can have terms with a rational coefficient and terms with a floating point coefficient. In addition the rule is that a PolyRatFun cannot have floating point numbers in its arguments. This would cause too many problems, even though one would expect this to help a lot with efficiency in some programs.

```
#startfloat 1000
Format 72;
CF f;
S  x,y;
L   F = f(x/3+y/5);
Print;
.sort

F =
   f(1/5*y + 1/3*x);

ToFloat;
Print;
.sort
```

```
F =
   1.0e+00*f(1/5*y + 1/3*x);

 Argument f;
     ToFloat;
 EndArgument;
 Print;
 .end

F =
   1.0e+00*f(2.0e-01*y +
   3.333333333333333333333333333333333333333333333333333333333333\
   333333333333333333333333333333333333333333333333333333333333\
   333333333333333333333333333333333333333333333333333333333333\
   333333333333333333333333333333333333333333333333333333333333\
   33333333333333333333333333333333333333333e-01*x);
```

We need two new routines for adding coefficients:

1. AddWithFloat for SplitMerge which sorts by pointer.
2. MergeWithFloat for MergePatches etc which keeps terms as much as possible in their original location.

The routines start out the same, because AT.SortFloatMode, set in Compare1, tells more or less what should be done. The difference is in where we leave the result.

In the future we may want to add an optional feature that makes the result zero if the sum comes within a certain distance of the numerical accuracy of the floating point numbers, like for instance 5% of the number of bits. There should not be a need to carry many terms around of which the coefficients are infinitesimal. This may not be very satisfactory for mathematicians, but when one considers problems of numerical stability, one should always run either with very high precision or with two different precisions.

With all of the above, we still miss a number of things: the optimizations cannot work with floating point numbers, and, as mentioned before, PolyRatFun cannot deal with them either. Optimizations with floating point numbers could be rather tricky, and one may wonder what extra problems it would introduce.

The next step is something that has been in preparation for a long time: implementation of a number of standard functions and constants. Of course we would like those also to be to arbitrary precision. The GMP library does not provided for that, but the MPFR library does. It provides the standard numerical functions that one will also find in Fortran and C. Unfortunately the internal notations of the two libraries are not exactly identical. The differences deal for a big part with rounding, but it means that occasionally we have to convert. This is not very complicated, and the structure/naming of the variables is rather similar. All dealings with the MPFR library have been collected in the file evaluate.c. Of course functions like sin_, cos_ have special cases for which we can provide an immediate and exact answer. This is all treated in the routine EvaluateFun. If the addition of new functions is required, it should be done in the same way as done inside this routine.

At this point we still miss functions that would be very useful for particle physics programs. Of course it would be an open-ended project to add all functions that are used in the modern literature, because frequently new functions are introduced. At the moment only the MZV's and the Euler sums have been implemented. The constants ee_ (for the number e) and em_ (for the Euler-Mascarponi constant) should still be added. The nuber pi_ is available, but one should not use FORM to compute it to millions or billions of digits. For that one can find dedicated programs on the internet.

One extra class of functions we would like to have consists of varieties of the Multiple Zeta Values. In first instance the MZV's and the Euler sums, but at a later stage sums involving different roots of unity as well. The paper by Borwein, Bradley, Broadhurst and Lisonek (Special values of Multiple Polylogarithms) provides information about how to program these functions as a combination of nested sums and it allows the determination of the depth of the sums. Because these sums can be done best, using up to w intermediate arrays, in which w is the maximum weight of the sums, and n is their length, which is related to the number of binary digits in the floating point numbers, we allocate these arrays when setting up the floating point numbers.

The evaluation of the MZV's and Euler sums requires some extra care, because they involve nested sums in which the maximum number of nestings is equal to the weight, and each step in the sums gains only one bit in accuracy. Hence, naively we would have at a weight 20 MZV with 5000 bits of accuracy $\mathcal{O}(5000^{20})$ steps. This would be prohibitive. The solution is to tabulate the innermost sum. Then use this table to make a table of the next sum. After this we do no longer need the innermost sum, and we can use this space for the table of the next sum. Etc. Now we only need $\mathcal{O}(5000 \times 20)$ steps. In principle one can gain a little bit more by keeping the tables for a number of the inner sums but this would require much space. In practise, we only keep very few sums that occur very frequently. The result is a very efficient routine which can compete with the 'best in the business'. Because the sums we need are all expansions in a value $x = 1/2$, internally we have a function that evaluates them with the above algorithms, and this function can also be accessed externally as mzvhalf_.

Together this involves the following routines:

```
#[ MZV :
    ## SimpleDelta :
    ## SimpleDeltaC :
    ## SingleTable :
    ## DoubleTable :
    ## EndTable :
    ## deltaMZV :
    ## deltaEuler :
    ## deltaEulerC :
    ## CalculateMZVhalf :
    ## CalculateMZV :
    ## CalculateEuler :
    ## ExpandMZV :
    ## ExpandEuler :
    ## EvaluateEuler :
#] MZV :
```

In principle one can use the BBBL paper also for working out the sums over an alphabet of different roots of unity. This has been done by Oliver Schnetz to construct tables for those MZV's, because they may be needed in future loop integrals, but FORM does not have them (yet?).

It would be nice to have a general Hpl function to arbitrary precision, but that is at the moment only in the contemplation stages. It seems like a good project for someone who wants to familiarise themselves with the topic.