19 Multiple threads

In 1991 I visited FNAL and talked with George Hockney about using the Fermilab machine for running FORM making use of multiple processors. In a week we got something running that did a rather simple test 100+ times faster than a single processor on the 257 processors of this Fermilab machine. Unfortunately this project could not be continued, but it helped me thinking in terms of parallelization. Then, in the mid 1990's, Hans Kühn asked me whether I was willing to collaborate on making a version of FORM that would run on several computers simultaneously. This did require different thinking, because in different computers with communication going via a network the requirements are different from those of a machine with multiple cores and shared memory. But with hard work from mainly Albert Retey and Denny Fliegner, under the supervision of Hans Staudenmaier, they did get it to work and this version was called ParFORM. The main problem with ParFORM is that most people do not have a farm of computers that is suitable for it.

In the years after more and more people obtained computers and workstations that were equipped with more than one core and these cores were sharing the main memory and the same disks. Hence TFORM was born to make use of this (2005). TFORM makes use of a standard that has been in use on most computers by now: the POSIX standard for parallel processing. The result is that it is easily portable and because there are by now many computers with a large number of cores, most of the heavy FORM running is done with the use of TFORM.

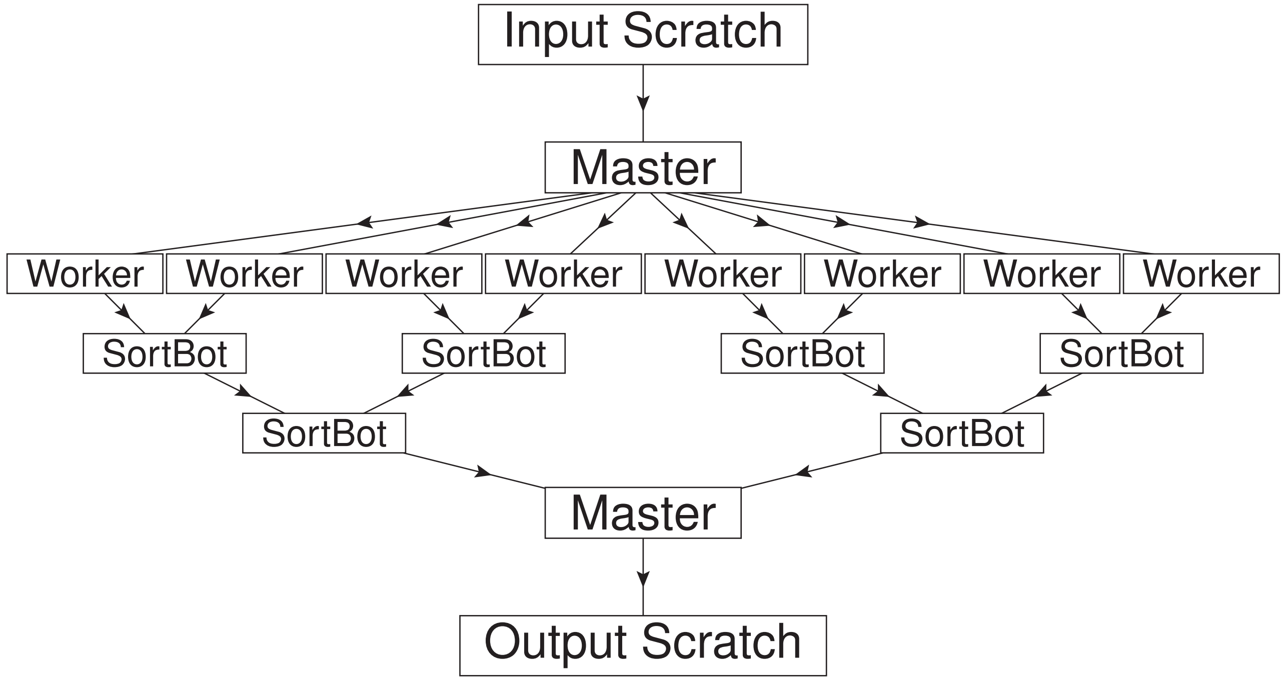
Although the parallelization model of TFORM is completely different from the way it was done on the Fermilab machine, it benefitted very much from what was learned there. It had taught me to think in terms of core specific data, something that was of course also needed for ParFORM, but in a different way. The first step was the split of the global data struct A into two parts when one needs TFORM. But this should only be noticeable by means of macro's to avoid having separate sources. As a result the private substructs N,R,T became part of a struct *B for TFORM while they remain part of A for the sequential version of FORM. To remind you we have

```
#ifdef WITHPTHREADS
    #define AT B->T
#else
    #define AT A.T
#endif
```

Each worker should have its own copy of *B with its private data and because the memory is shared, it does not matter on which core each worker is running. The scheduler will take care of that. Hence the model is:

- We have a master process. It makes the allocations and does the work that has to be done when only one process can be active.
- We have N workers, each of which gets tasks assigned by the master and eventually gives their finished work back to the master.

- We have N-2 sortbots that help during the later stages of the sorting to avoid that the master has to do too much work in the EndSort stages of the sort.



To keep things more or less coherent, all the TFORM particular code resides in the file threads.c. In the other files there are occasional little pieces of code like in variable.h

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

or in structs.h (quite a few more occurrences as well)

```
typedef struct DoLlArS {
    WORD    *where;              /* A pointer(!) to the object */
    FACDOLLAR *factors;          /* an array of factors. nfactors elements */
#ifdef WITHPTHREADS
    pthread_mutex_t pthreadslockread;
    pthread_mutex_t pthreadslockwrite;
#endif
    LONG    size;                /* The number of words */
    LONG    name;
    WORD    type;
    WORD    node;
    WORD    index;
    WORD    zero;
    WORD    numdummies;
    WORD    nfactors;
#ifdef WITHPTHREADS
    PADPOINTER(2,0,6,sizeof(pthread_mutex_t)*2);
#else
    PADPOINTER(2,0,6,0);
#endif
} *DOLLARS;
```

In the DOLLARS definition TForm needs some locks when results have to be written to the central administration as in:

```
#$maxpow = 0;
if ( count(x,1) > $maxpow ) $maxpow = count_(x,1);
moduleoption max,$maxpow;
.sort
```

Without the module option TForm cannot garantee that indeed the maximum power ends up in $maxpow. But the moduleoption makes it such that by placing locks it is possible to make sure that no other worker is trying to change the content of the variable between that we read it to see whether we have to replace it and that we actually do replace it. And to be completely safe the order is

1. Read the value.
2. If we have a better value we place a lock, both for reading and for writing.
3. Check whether we still need to replace it!
4. If we still want to replace it, we can do that now.
5. Release the locks.

This kind of thinking is needed when constructing a proper multi-core program. There are more than 300 of such WITHPTHREADS pieces of code in the sources of Form, but for most additions and service one does not have to worry too much about them. The only places that do need attention are the calling of subroutines that read or write data from the *B private structs, meaning that they use variables in AN, AR or AT. In that case one needs to include the macro's BHEAD in the call to those routines and PHEAD in their definitions. Let us now have a look at the main pieces of the TForm program.

When the program starts, we have to start a few features that deal with the identities of the threads. This is done in the routine BeginIdentities. Next the program determines how many workers there must be. After that it can allocate those workers and a few crucial arrays. This is done in the routine StartAllThreads. The various threads are initialized. This means that buffers are allocated for them. We use that the sort buffers for the workers are 1/N times the size of the sort buffers of the master. In the case that there are sortbots, those also obtain 1/N times the size of the corresponding buffer of the master. This does not hold for the size of their workspace. That size is the same for both the master and the workers.

Once the buffers have been set up the workers are ready to start working. The master will read input expressions from the scratch file and will have to distribute the terms over the workers. This requires care. If terms are sent one by one there is quite some overhead in the communication. Each signal takes time. Hence we work with ‘buckets’. Typically a bucket size of 500 terms makes that there is not too much overhead. Of course when there are fewer than $500N$ terms each bucket will get a fair share. This bucket system is run from the routine ThreadsProcessor. This routine maintains an array of 2N buckets which are filled and when a worker is available, the address of a full bucket is sent to it. When a worker is done its bucket is empty, so when all workers are working on their bucket the master can fill empty buckets. An interesting situation occurs when there are no more terms but some workers are still working because they happen to have some ‘difficult’ terms. When workers are finished and there are no more terms the master looks in the buckets of the workers and takes the bucket that is still fullest and ‘steals’ the terms that are not being worked at to distribute them over the empty buckets and hand those to the idle workers. This can be quite important because very often most of the difficult terms are grouped together and may end up in the same bucket.

The signals that the workers can receive and the action that the workers take when being notified can all be found in the routine RunThread. There are currently 12 signals, each in their own fold inside a switch:

```
    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,i
                MUNLOCK(ErrorMessageLock);
                Terminate(-1);
                break;
        }
    }
```

The HIGHERLEVELGENERATION concerns the model that was made for the Fermilab computer. In that model every branch in the evaluation tree looked whether it could give some task to an idle worker. If it had for instance

```
    id  x = a+b+c+d;
```

it would look whether it could give the a, the b and/or the c to other workers before doing it by itself. Only the last term could never be given to another worker, because otherwise one would get tasks in orbit, which resembles a burocracy. Hence by the principle of that you cannot pass the last task, we jokingly claimed that we had solved burocracy. Unfortunately under POSIX the passing of such small tasks and specially deep in the tree, gives way too much overhead and hence the system did not work well in TFORM. But some of the code is still there.

During the processing of the terms there are not many things that are thread specific. This brings us to when there are no more terms and the workers have to sort their terms. Each worker has its own sort system with a small buffer, a large buffer, a sort file, if necessary a stage4 file but instead of the output scratch file the output by means of the routine PutOut at that stage is to a buffer. Originally, if there would be N workers there would be one giant buffer, divided into N subbuffers. Each of those N subbuffers would again be divided into 10 equal pieces. The workers would be filling those 10 pieces and the master would be merging, working on one of those 10 pieces per worker, using high and low watermarks. This means that the workers could be filling at the same time that the master would be reading. With the advent of the sortbots things are a bit different. Now there are extra workers called sortbots that are only active during this final merge, but each worker merges two outputs into a single output. The final merge of two outputs is done by the master who writes the output to the output scratch file. This insertion of the sortbots speeds up the sorting significantly. After all the two major bottlenecks in TFORM are the reading of the input with the division of the terms over the workers and the inverse tree at the end of the sorting with the writing to the file by the master. It is important to make this bottleneck as insignificant as possible and the processing by the workers the major part of the program.

If one has very many expressions, none of which excessively large, one can use the InParallel statement. In that case the master doesn't divide the terms over the workers, but it divides the expressions over the workers. This means that for each expression basically one message needs to be sent. The worker deals with the complete expression, writes it to its own scratch file and then asks for a lock to be able to write the finished product as quickly as possible to the main output scratch file. It does means however that the order of the expressions in the scratch system is no longer a fixed order.

Another way to gain speed is when we use the bracketindex (bracket+). In that case the master tells the worker to pick up complete brackets. This has two advantages. The first is in the reading. This is now done by the workers. The second is shown in the next example

```
AntiBracket+,y,x1,x2;
.sort
id y = x1+x2;
* possibly another antibracket statement to prepare for the next module
.sort
```

What happens now is that after the first sort, the brackets contain all occurrences of y, x1, x2 which means that the results of the id-statement make that all possible additions or cancellations between terms take place inside the same worker and as early as possible during the sort. Most likely already inside the small buffer. The result is fewer compares per term and less space needed in the buffers and files. Statements like the antibracket statement, the bracket index and the distribution over the workers by bracket were invented when memory, disk space and CPU time were at their limits for a given calculation.

The above model for distributing terms over the workers has one very weak point. In the module in which an expression is defined the RHS is often only a single term or subexpression. That means that in that module only a single worker can be employed for that expression. This is shown in the following example in which I have edited out the variables:

```
On codes;
```

```
    L   F1 = (a+b)^3;
    L   F2 = a-b;
    .sort
 Right Hand Sides:
   8  1  4  20  1  1  1  3  8  1  4  21  1  1  1  3

   9  6  5  1  3  0  1  1  3

   8  1  4  20  1  1  1  3  8  1  4  21  1  1  1  4294967293

    L   F3 = F1;
    .end
 Right Hand Sides:
   9  5  5  0  1  0  1  1  3
```

In the first and third cases the RHS is a single term. In the first case a subexpression(6), and in the third case an expression(5). In one of the exercises you will be asked to study how the third case can be dealt with. The first case is much more difficult. Maybe some creative use of the HIGHERLEVELGENERATION possibility can help there to make the distribution of terms one level from the ground level, but because that system has not been tested very thoroughly, it might be dangerous. The regular workaround is to make sure that in the defining module only an intermediate number of terms are generated and that the real explosion in the number of terms happens after a .sort.

Sometimes people ask what is the best setup for FORM or TFORM. And what is a good computer for these programs? It should be clear that much memory is very helpful, because use of the disks can slow things down a bit when the expressions become very large. This is in particular the case when the sorting will have to go into stage4. If one has 32 workers and each is sorting from the sort file to the stage4 file, one can obtain quite a traffic jam at the disk. Of course SSD disks improve the situation very much, The best computer I have at Nikhef has 64 cores, 1 Tbyte RAM, 20 Tbytes of SSD and 6 Tbytes of regular disks. The SSD is in principle completely empty, because it is reserved for the temporary files of FORM. All work gets done from directories on the regular disk. In some places like DESY it used to be that after a given amount of time one would loose the AFS token for remote file access. The result was that if the executable of FORM was run via this remote link, the program would stop as soon as the token was lost. Hence it is always best to have the executable and all relevant files on the local disk of the computer on which one is running. Another thing to take into account is that FORM does not always know how much memory it will need. Some buffers may grow during execution. And some allocated space may not be used entirely. If one is running on a computer that has been set up for batch jobs, one has to know how much memory is needed and one cannot exceed this. There is no swap space on batch computers. The FORM computers at Nikhef have been set up with as much swap space as there is memory. This makes that the running proceeds usually very smoothly, although computer centers usually do not like such a setup. Of course one has to keep programs like Mathematica away from such computers, because when Mathematica starts using the swap space, the computer will more or less die for all users on it.

19.1 Pseudo-efficiency

When running a program with TFORM, we call the efficiency of the parallelization the total time the program was running divided by the time the program takes with the regular sequential version of FORM. If we run for instance on 8 workers on my (very good) laptop we obtain

```
    TFORM 5.0Beta (Apr  5 2023) 8 workers            Run: Sun Apr  9 11:04:12 2023
    #: smallsize 100M
    #: largesize 500M
    #: termsinsmall 2M
    #: SortIOsize 200K
    #: ScratchSize 116M
    #ifndef ‘NUM’
    #define NUM "14"
    #endif
    #message Trace of ‘NUM’ gamma matrices in 4 dimensions
~~~Trace of 14 gamma matrices in 4 dimensions
    S x,j;
    CF f;
    L FF = sum_(j,1,1000,f(j));
    .sort
```

```
Time =       0.00 sec    Generated terms =       1000
              FF         Terms in output =       1000
                         Bytes used      =      24016
    I m1,...,m‘NUM’;
    id f(x?) = 1;
    Multiply G_(1,m1,...,m‘NUM’);
    Trace4,1;
    .end
                .
                .
             Thread 8 reporting
Time =       1.73 sec    Generated terms =    3886677
              FF         Terms in thread =      26931
                         Bytes used      =    2154484

Time =       0.00 sec    Generated terms =   31599000
              FF         Terms in output =      26931
                         Bytes used      =    1025268
  0.00 sec + 13.88 sec: 13.88 sec out of 1.77 sec
```

while on Form we obtain for the same program the last line is

```
  10.99 sec out of 11.00 sec
```

This gives an efficiency of $11.00/1.77 = 6.21$. This is less than 8 due to the overhead in the parallelization.

Unfortunately we do not always have the corresponding FORM run to compare with and we have only the results from the TFORM run, because that is why we use TFORM. Hence we have only the total amount of CPU time that TFORM spent. We define the pseudo-efficiency as this CPU time divided by the real time: $13.88/1.77 = 7.84$ which in this case means that the parallelization was near perfect in terms of CPU time.

When running TFORM (or ParFORM of course) we would like to write the program in such a way that the pseudo-efficiency is as high as possible. On N workers that would mean as close as possible to N. Actually, when we spy with top, occasionally the percentage of CPU use exceeds N times 100% when all the workers are occupied and the master is filling buckets, or when all workers are still finalizing their sorting and the sortbots are already processing the initial parts of their output. This is however very rare. I have never seen a pseudo-efficiency above N.

The question is now how to maximize the pseudo-efficiency. Clearly it means that we should keep the workers busy and hence that the modules should not be very short, unless during the early stages of the sorting most terms add or cancel. Heavy manipulations of function arguments will help as well, because for instance

```
    Argument f;
        id a = b+c;
    EndArgument;
```

will sort the argument(s) of f after the substitution and hence the number of terms at the ground level does not increase because of this. Hence a whole sequence of Argument/EndArgument environments makes usually for rather efficient programs, because it leaves a relatively small amount of work for the master.

One thing that takes the pseudo-efficiency down considerably is to have only one term (or very few terms) in the input. The distribution of terms, as explained before, takes only place with the input from the scratchfile. This explains why we have an extra .sort in the above example: we first create 1000 terms, and in the next module the actual work is done. If we were to omit that .sort the result would be

```
    #: smallsize 100M
    #: largesize 500M
    #: termsinsmall 2M
    #: SortIOsize 200K
    #: ScratchSize 116M
    #ifndef `NUM'
    #define NUM "14"
    #endif
    #message Trace of `NUM' gamma matrices in 4 dimensions
~~~Trace of 14 gamma matrices in 4 dimensions
    S   x,j;
    CF  f;
    I   m1,...,m`NUM';
    L   FF = sum_(j,1,1000,f(j));
```

```
    *.sort
    id  f(x?) = 1;
    Multiply G_(1,m1,...,m`NUM');
    Trace4,1;
    .end
                    .
                    .
            Thread 3 reporting
Time =       11.15 sec    Generated terms =   31599000
              FF          Terms in thread =      26931
                          Bytes used      =    2154484

Time =        0.00 sec    Generated terms =   31599000
              FF          Terms in output =      26931
                          Bytes used      =    1025268
  0.00 sec + 11.15 sec: 11.15 sec out of 11.15 sec
```

and close inspecion reveals that all is executed in thread 3. It is one of the exercises to figure out what we can do about some of the cases in which this problem occurs.

20 Dollar variables

Dollar variables must be able to represent a variety of objects. They are not declared like the algebraic variables. Hence the system has to be able to determine their properties and, if needed, to convert from one type to another. For each dollar variable there is a struct, given by

```
typedef struct DoLlArS {
    WORD    *where;             /* A pointer(!) to the object */
    FACDOLLAR *factors;         /* an array of factors. nfactors elements
*/
#ifdef WITHPTHREADS
    pthread_mutex_t pthreadslockread;
    pthread_mutex_t pthreadslockwrite;
#endif
    LONG    size;               /* The number of words */
    LONG    name;
    WORD    type;
    WORD    node;
    WORD    index;
    WORD    zero;
    WORD    numdummies;
    WORD    nfactors;
#ifdef WITHPTHREADS
    PADPOINTER(2,0,6,sizeof(pthread_mutex_t)*2);
#else
    PADPOINTER(2,0,6,0);
#endif
} *DOLLARS;
```

and FACDOLLAR is defined by

```
typedef struct FaCdOlLaR {
    WORD    *where;             /* A pointer(!) to the content */
    LONG    size;
    WORD    type;               /* Type can be DOLNUMBER or DOLTERMS */
    WORD    value;              /* in case it is a (short) number */
    PADPOINTER(1,0,2,0);
} FACDOLLAR;
```

Contrary to what we try to avoid with terms in the main algebraic expressions, here contents are stored in fixed memory locations and hence we keep pointers to them. This means that dollar expressions should not be very big. And because dollar expressions can be factorized (with either the #factdollar instruction or the factdollar statement) those factors can be stored in the array factors. Often a dollar variable is a rather simple object though, in which case we do not allocate memory in the array 'where'. The name of the dollar is inside

AC.dollarnames[dol.name] when dol is its struct. A dollar variable can be added with the routine AddDollar in names.c:

```
    int AddDollar(UBYTE *name, WORD type, WORD *start, LONG size)
```

Here name is the name, type the type, start the beginning of the contents if they are to be stored in 'where', otherwise zero, and size the size of these contents in WORDs. If start has a value (an address) and size is greater than zero an allocation will be made. Otherwise the default value will be zero. For this zero there is a default buffer AM.dollarzero. This buffer should never be changed, because 'where' will be pointing at it when the value is zero, and so will all other dollar variables that contain the value zero. The routine AddDollar creates also an entry in the special compiler buffer for dollar variables (AM.dbufnum).

The type of what the dollar contains is given in ‘type’. This can be one of

```
#define DOLUNDEFINED 0
#define DOLNUMBER 1
#define DOLARGUMENT 2
#define DOLSUBTERM 3
#define DOLTERMS 4
#define DOLWILDARGS 5
#define DOLINDEX 6
#define DOLZERO 7
```

When the type is DOLTERMS there must be an allocation for ‘where’ and the content of the indicated memory is an expression, i.e. a sequence of terms terminated by a zero. Also some of the other types need their values stored in ‘where’. One should be quite careful to release the memory in ‘where’ when the buffer has to be reallocated. This is seen in the routine WildDollars which is for storing dollar values as in

```
id f(a?$a1,?x$x) = ...
```

in which $x will be of the type DOLWILDARGS and if a is a symbol, we have to construct a term with a single symbol. This will need 9 WORDs (including the trailing zero) and we use the type DOLTERMS. If the wildcard becomes an index we just keep it in the element ‘index’ of the DOLLARS struct and the type is DOLINDEX. The element ‘numdummies’ needs some attention. This is for when a dollar contains contracted indices that have been summed over. Those dummy indices might cause problems when the dollar is multiplied with a term that also contains dummy indices. In that case the internal numbers of those dummy indices would have to be changed. The same holds when we have powers of the dollar variable.

The array of dollar variables is kept in the variable AP.DollarList. We access it by means of the macro's

```
#define Dollars ((DOLLARS)(AP.DollarList.lijst))
#define NumDollars AP.DollarList.num
```

Hence the dollar variable n is accessed with `Dollars[n]`. When a new dollar variable needs to be introduced this is done with the routine FromVarList in tools.c as in

```
DOLLARS dol = (DOLLARS)FromVarList(&AP.DollarList);
```

This routine automatically doubles the size of the array AP.DollarList is there is not enough room.

The file dollars.c contains quite a few routines to enter dollar variables. Basically a routine for each of the popular types of variables. and there is also a routine to convert the contents of a dollar variable to a character string. This is for use as a preprocessor variable, or for printing. This string 'value' is purely cosmetic. Dollar variables do not take real string values because in that case they would cause serious problems during the execution phase.

When we have TForm and there are dollar variables involved, the first reaction is that if, during the execution of the module, any dollar variable is changed, the module cannot run in parallel. But there is a method to override this with the moduleoption statement. The three options max, min and sum can be resolved easily with locks. The local option however needs more care. For this we have the struct (also used for the other module options)

```
typedef struct MoDoPtDoLlArS {
#ifdef WITHPTHREADS
    DOLLARS dstruct;    /* If local dollar: list of DOLLARS for each thread */
#endif
    WORD    number;
    WORD    type;
#ifdef WITHPTHREADS
    PADPOINTER(0,0,2,0);
#endif
} MODOPTDOLLAR;
```

An array of these structs

```
#define ModOptdollars ((MODOPTDOLLAR *)(AC.ModOptDolList.lijst))
#define NumModOptdollars AC.ModOptDolList.num
#define PotModdollars ((WORD *)(AC.PotModDolList.lijst))
#define NumPotModdollars AC.PotModDolList.num
```

keeps all dollar variables mention in the module options and PotModDollars is the array of potentially modified dollars that is constructed by the compiler. If there are dollars in PotModDollars that do not exist in ModOpt-Dollars, the module runs only on the master. If a dollar has been declared as local, 'dstruct' will contain an array of DOLLARS, one for each worker and each dollar will be a copy of the dollar in the master as it is at the moment before the execution of the module. This means that changes made by the preprocessor will be part of these copies. Then, when the value of a dollar is needed during execution, Form sees that the dollar is in this array dstruct and replaces the DOLLARS struct for that variable by the local DOLLARS struct. At the end of the module the whole ModOptDollars array is cleared and all local values are lost. The master still has the value it had at the start of the execution of the module.

In ParForm the above is rather different. ParForm broadcasts the value of all dollars to all workers and gathers in their new values at the end of the module. Conditions for executing the module are identical, but now the local copies are the easy part and the other three options require a bit of work during the gathering. It is possible to write code that will give different results on TForm and ParForm, because TForm puts for instance a maximum directly into the masters administration causing this maximum to be used by all workers for the remainder of the module. In ParForm the other workers may never be confronted with this 'local' maximum. But, because in parallel processing the order of the processing of terms is not fixed anyway, such programs would be unsafe anyway.

21 Wildcards

One of the powerful, but also complicated features of FORM is the wildcarding system which makes the pattern matcher rather versatile. Let us have a closer look of what is expected of a wildcarding system.

1. It needs a way to mark wildcards in the LHS of a substitution. We do need a distinction between fixed variables and generic variables. In different languages this is done differently. In FORM the addition of a questionmark to the name of a wildcard was selected. This seems rather natural because we do not yet know what value it will obtain.

2. There needs to be a way to restrict the values that a wildcard can obtain. In FORM this is done by attaching either the name of a set or the dynamic definition of a set. Usually the last method is more readable.

3. In case there is no match, the pattern matcher must be capable to be certain about this. If a match is possible, the pattern matcher should find it, preferably in as little time as possible. In the section about the pattern matcher we have already seen that there are cases in which FORM cannot handle this currently. There are cases in which it can take very much time.

4. Once a match is found, the information obtained should be applied consistently to the RHS of the substitution. As FORM shows, it is not necessary to have a special notation for the corresponding variables in the RHS. One could argue for

   ```
   id x^n?!{-1,} = x^(n?+1)/(n?+1);
   ```

and make a distinction between `n?` and `n`, but that would be asking for errors like forgotten questionmarks. And in addition one might have to declare `n?` separately to make sure about its type.

5. If one has different data types, wildcarding should be possible for all relevant data types.

6. It should be possible to pick up ranges of arguments of functions.

Probably there are more features that could be handy.

Let us now have a look how this is implemented in FORM. When a LHS is read by the compiler and a questionmark is encountered the variable is put in a list and, depending on the type, an offset is added to the number of the variable. This offset puts it outside the range of regular variable numbers, making it easily recognizable as a wildcard. This offset is,

Symbol 2×MAXPOWER. In the 64 bits version this should be 10^9.

Index WILDOFFSET. In the 64 bits version this is 400000100.

Vector WILDOFFSET. In the 64 bits version this is 400000100. Because vectors have a negative number when used inside terms, the value of AM.OffsetVector should be of the order of −2×WILDOFFSET.

Function WILDOFFSET.

When the LHS of a statement is compiled the compiler builds a prototype of a subexpression subterm for the RHS. In this prototype the information about the wildcards is stored in units of 4 WORDs. These units start with the type of wildcard, the size (4), the variable number and space for its substitution during execution. Example:

```
    S x,n;
    L   F1 = (x+1)^3;
    .sort
    On codes;
    id  x^n? = x^(n+1)/(n+1);
    Print;
    .end
     Left Hand Sides:
       1  21  2  4294967295  0  0  6  9  2  1  0  2  4  21  21  5  1  4  20
       1000000021  0
     Right Hand Sides:
       8  1  4  21  1  1  1  3  4  1  1  3

       42  20  20  1  4294967295  20  15  1  13  6  9  1  1  0  2  4  21  21  1  1
         3  21  18  1  15  1  13  6  9  1  1  0  2  4  21  21  1  1  3  1  1  3
```

The LHS is given by the pieces

```
       1  21  2  4294967295  0  0
          6  9  2  1  0  2  4  21  21
          5  1  4  20  1000000021  0
```

The first line is telling that we have a regular id statement. The last three numbers deal with options. We will not consider those. The second line gives the subexpression prototype. It tells that the RHS is in the second rhs in compiler buffer zero and it has one power. The four last WORDs are the ones that we need: the code is SYMTOSYM, indicating a symbol that, for now, will be replaced by a symbol, and the symbol is number 21 which is n. The last number indicates that for now we do as if it will be replaced by n. The third line is the actual LHS, which has 5 WORDs and we see a subterm of type SYMBOL with in it symbol x (20) to the power 'n?'.

Next we will have a look at what the pattern matcher does with this. The code for matching symbols and their powers is in the file findpat.c. In this case, because of the wildcard, the option on the id statement is 2 which is SUBONCE and hence routine FindOnce is called (the calling code is in TestMatch in pattern.c). As one can see, the commentary dates a bit. It is from before the first release of FORM. In the subfold SYMBOLS we find the main loop

```
    if ( *m == *t && t < xstop ) {
        nt = t[1];
        mt = m[1];
        if ( ( mt > 0 && mt <= nt ) ||
             ( mt < 0 && mt >= nt ) ) { m += 2; t += 2; }
        else if ( mt >= 2*MAXPOWER ) goto OnceL2;
        else if ( mt <= -2*MAXPOWER ) {
            nt = -nt;
            mt = -mt;
```

```
OnceL2:     mt -= 2*MAXPOWER;
            if ( ( ch = CheckWild(BHEAD mt,SYMTONUM,nt,&newval3) ) != 0 ) {
                if ( ch > 1 ) return(0);
                if ( AN.oldtype != SYMTONUM ) return(0);
                if ( AN.oldvalue <= 0 ) {
                    if ( nt < AN.oldvalue ) nt = AN.oldvalue;
                    else {
                        if ( *AN.MaskPointer == 2 ) return(0);
                        if ( nt > 0 ) nt = 0;
                    }
                }
                if ( AN.oldvalue >= 0 ) {
                    if ( nt > AN.oldvalue ) nt = AN.oldvalue;
                    else {
                        if ( *AN.MaskPointer == 2 ) return(0);
                        if ( nt < 0 ) nt = 0;
                    }
                }
            }
            AddWild(BHEAD mt,SYMTONUM,nt);
            m += 2;
            t += 2;
        }
        else {
            *p++ = *t++; *p++ = *t++; n += 2;
        }
    }
```

When we have pattern matching usually the main pointer in the pattern is m while the main pointer in the term is t. Here mt is the power of the symbol which is our 'n?' while nt is the value 3. The call to CheckWild (in wildcard.c) asks whether this wildcard assignment can be made. With symbols there is a certain flexibility, because symbols can match either a symbol (SYMTOSYM) or a number (SYMTONUM) or a whole subexpression (SYMTOSUB). We have made a copy of the original wildcard information, hence before the call to CheckWild the value of AN.oldtype is SYMTOSYM, but CheckWild can see that there was no assignment yet and hence it can overwrite AN.oldtype into SYMTONUM and it approves of the match. This causes FORM to call the routine AddWild (also in wildcard.c) to make an assignment in the list of wildcards. After this it continues to the next symbol if there is any. Although things are more complicated when the x would also be wildcarded, we will skip that here. It might involve some backtracking and because of that at each stage we have to be able to go back to previous assignments. In the case of the example there is only a single symbol and hence we are done now and the routine will give the OK for the match with the wildcard assignments stored by AddWild. After this the pattern has to be taken out. This is done in the routine Substitute in the file pattern.c. The code there is rather similar, but now we only have to check whether an assignment is allowed with CheckWild and no further backtracking is needed. The pattern is taken out and replaced by a subexpression subterm in which the wildcards have their proper values in case they have to be passed on to deeper subexpressions.

The actual substitution in the RHS happens when the RHS terms are inserted in either InsertTerm and FiniTerm in the file proces.c. These routines call the routine WildFill in the file wildcards.c.

The above is the basic procedure for all wildcards, although the more complicated the wildcards or the terms the more cases have to be considered. The best example is the routine MatchFunction in the file function.c. We will not look at the particular code here, but it is full of special cases, backtracking for when CheckWild disapproves of a match and no further matches are possible, etc.

Pattern matching can be very complicated when one considers topological structures as when one wants to determine the topology of a Feynman diagram. The vertices are symmetric functions v with the edges represented by contracted indices between the vertices. If we want to test whether a given diagram would have this topology (and what would then be the assignments of the edge and vertex numbers) the pattern would read

```
id  v(i0?,i1?,i2?)*v(i0?,i3?,i4?)*v(i1?,i5?,i6?)*v(i2?,i7?,i8?)*
v(i3?,i9?,i10?)*v(i4?,i11?,i12?)*v(i5?,i13?,i14?)*v(i6?,i15?,i16?)*
v(i7?,i17?,i18?)*v(i8?,i19?,i20?)*v(i9?,i13?,i17?)*v(i10?,i15?,i19?)*
v(i11?,i14?,i20?)*v(i12?,i16?,i18?) = success;
```

If a diagram matches, Form will usually find it rather quickly, but the problem is when a diagram does not match. Form may well try all possible assignments of the indices before it concludes that this does not match. A human would of course try to look for different criteria and see rather quickly that there is no way that this can match. For cases like this we have the file smart.c that tries to intercept a number of clear cases. This file is however far from complete and maybe it never will be. But if one is faced with very slow patterns that can be resolved by being a lot smarter than FORM, please consider whether it is possible to make additions to this file.

The prototype becomes more complicated when sets are restricting the value of the wildcards as in

```
    Symbols x,y,a,b,c,d;
    Set ab:a,b;
    CFunction f;
    Local F = f(a,b)+f(d,b)+f(b,c);
    .sort
    On codes;
    id  f(x?ab,y?!ab) = f(y,x);
    Print;
    .end
     Sets
       pos_(0): integers > 0
       pos0_(1): integers >= 0
       neg_(2): integers < 0
       neg0_(3): integers <= 0
       even_(4): even integers
       odd_(5): odd integers
```

```
  int_(6): all integers
  symbol_(7): only symbols
  fixed_(8): fixed indices
  index_(9): all indices
  number_(10): all rationals
  dummyindices_(11): dummy indices
  vector_(12): only vectors
  ab(13): a b
Expressions
  F(local)(0)
Expressions to be printed
  F
Dollar variables
  $$(0)
Left Hand Sides:
  1  36  4  4294967295  0  0  6  21  1  1  0  2  4  20  20  14  4  13
  3894967196  2  4  21  21  14  4  800000213  3894967196  8  150  7  0
  4294967295  1000000020  4294967295  1000000021  0
Right Hand Sides:
  11  150  7  1  4294967295  21  4294967295  20  1  1  3
```

The interesting part is the subexpression prototype:

```
  6  21   1   1   0
     2   4   20   20   14   4   13   3894967196
     2   4   21   21   14   4   800000213   3894967196
```

Each wildcard is followed by another entry that starts with the number 14. This 14 is the code for 'FROMSET' and in the first case we see set number 13 which is ab. In the second case we see actually set 13 + 2×WILDOFFSET. This offset is to indicate the ! (not) operator. The last number in the FROMSET block is not relevant here. It can be used when we have patterns involving a?ab?xy if xy is another set. In the case of a pattern involving a?ab[n] we would get `15 4 13 26` assuming that 26 is the number of the symbol n. 15 is the code for SETTONUM. Yet another case is when we want to capture the value of a wildcard in a dollar variable:

```
    Symbols x,y,a,b,c,d,n;
    Set ab:a,b;
    CFunction f;
    Local F = f(a,b)+f(d,b)+f(b,c);
    .sort
    On Codes;
    id  f(x?ab[n],y?!ab$catch) = f(y,x)*f(n);
    Print;
    .sort
 Dollar variables
   $$(0) $catch(1)
 Dollar variables to be modified
   $catch
```

```
Left Hand Sides:
  1  44  4  4294967295  0  0  6  29  1  1  0  2  4  20  20  15  4  13  26  2
    4  21  21  14  4  800000213  3894967196  18  4  1  1  2  4  26  0  8  150
    7  0  4294967295  1000000020  4294967295  1000000021  0
Right Hand Sides:
  16  150  7  1  4294967295  21  4294967295  20  150  5  1  4294967295  26  1
    1  3

  F =
     f(a,b) + f(d,b) + f(2)*f(c,b);

   Off codes;
   #write "                $catch = `$catch'"
               $catch = c
   .end
```

and now the wildcard part has become

```
  2  4  20  20    15  4  13  26
     2  4  21  21    14  4  800000213  3894967196    18  4  1  1
  2  4  26  0
```

This is a bit harder to read. The first wildcard forces n to be wildcarded as well. The sorting of the wildcards has caused this one to come last. But it is the second wildcard we are interested in. It consist now of three entries. The second we have seen before as it concern the set. The third one 18 means LOADDOLLAR and indicates the capture into dollar number 1.

Next we need to know where all this action takes place. The set information can be checked in CheckWild. The harder part is how to deal with the dollar variable(s). The routine that does this is WildDollars in the file dollar.c. It is called from the routine TestMatch in pattern.c. This routine checks before anything else how many dollar variables have to be filled and keeps that number in the variable numdollars. Then after the matching is judged successful, and when this variable is not zero WildDollars will be called. The communication between the various routines is by a copy of the wildcard information in the variables AN.FullProto, AN.WildValue and AN.WildStop. These last variables are also used for making copies during the matching to allow the backtracking. Note that filling dollar variables this way is potentially unsafe when running either TForm or ParForm.

22 Messages and output

At various stages FORM has to print output and (error)messages. The regular output routines can be found in the file sch.c. They are relatively easy and the main complications are having the output formats tuned to different languages. Output is collected in a fixed size array, which is printed when its contents reach a given limit. Lately people have been suggesting that it would be nice if this fixed line limit could be removed. This should not be too difficult, because it would mean to just omit the linefeed that is automatically printed at the end of a line inside an expression and not printing the leading blank spaces at the start of the next line. Here I leave this as one of the exercises.

Normally the output expressions are written to stdout. When the -l option is used as in `form -l inputfile` the output is written both to stdout and to the .log file. But if in a print statement the option +f is used and there is a .log file, the output expression is only written to the .log file. This is of course all relatively simple.

The messages are in the file message.c. Some messages are very frequent and have their own routine, like MesWork for workspace overflow. The major routine however is MesPrint, which the FORM equivalent of prinf in the C language. Of course we need escape sequences that are different from those in computational languages. All these sequences are indicated in the commentary.

```
/*
    Kind of a printf function for simple messages.
    %a  array of size n WORDs (two parameters, first is int, second WORD *)
    %b  array of size n UBYTEs (two parameters, first is int, second UBYTE *)
```

```
%C  array of size n chars (two parameters, first is int, second char *)
%d  word;
%l  long;
%L  long long *;
%s  string;
%#i unsigned word filled
%#d word positioned
%#l long word positioned.
%#L long long word * positioned.
%#s string positioned.
%#p position in file.
%r  The current term in raw format (internal representation)
%t  The current term (AN.currentTerm)
%T  The current term (AN.currentTerm) with its sign
%w  Number of the thread(worker)
%$  The next $ in AN.listinprint
%x  hexadecimal. Takes 8 places. Mainly for debugging.
%%  %
%#  #
#   " ==> "
@   " ==> "   Preprocessor error
```

```
    &   ' --> '   Regular compiler error
    Each call is terminated with a new line.
    Put a % at the end of the string to suppress the new line.
*/
int MesPrint(const char *fmt, ... )
{
    GETIDENTITY
```

When, during execution, FORM encounters the `Print "...";` option the amount of work for it was relatively limited, because most of the options are passed on directly to MesPrint. This is the same with the #write instruction of the preprocessor. The weakest point in this approach is that MesPrint was originally designed for error messages which would always end with a linefeed. Hence writing a single line that is composed of several #write instructions is a bit non-standard. This can be repaired by making an escape sequence for adding an ASCII 0 (like `'\0'` in C) at the end of a line. The routine that actually writes the string (WriteString in the file tools.c) will strip this zero and not add a linefeed in such a case. This must still be done (one of the exercises).

The reason there are more than 2600 calls to Mesprint in the FORM sources is that the compiler part has to intercept very many types of errors, and even with this large number one could always argue that more is better. I always learned that meaningful error messages are one of the hard parts of a large computer program. If you make additions to FORM, please keep this in mind. It is very clear that currently FORM has messages during execution that are very hard to understand for non-experts. And also compiler messages sometimes leave the user searching for some time to find the particular typo. A simple example that most of us have run into is

```
  S   a,b
  F   A,B;
  L   Exp = A*B*A*B+(a+b)^2;
  Print;
  .end
```

```
 Exp =
    A^2*B^2 + b^2 + 2*a*b + a^2;
```

This program is grammatically correct, and hence FORM cannot detect that it is probably not what the user meant (Why `A^2*B^2` and not `A*B*A*B`?).

It becomes clearer with

```
    CF  F;
    S   a,b
    F   A,B;
ac.frm Line 2 --> F has been declared as a function already
    L   Exp = A*B*A*B+(a+b)^2+F(a-b);
    Print;
    .end
Program terminating at ac.frm Line 5 -->
```

It is still not perfectly clear, but at least there is a hint of what is wrong.

To create an extensive warning system about potentially dubious use would be a major undertaking for which most physicists do not have the time. And on top of that, many people turn off the warnings that Form does have (not so many).

23 Tables

Tables are functions with special properties. In the file structs.h we can find the struct for functions:

```
typedef struct FuNcTiOn {  /* Don’t change unless altering .sav too */
    TABLES  tabl;          /* Used if redefined as table. != 0 if function is a table */
    LONG    symminfo;      /* Info regarding symm properties offset in buffer */
    LONG    name;          /* Location in namebuffer of #NAMETREE */
    WORD    commute;       /* Commutation properties */
    WORD    complex;       /* Properties under complex conjugation */
    WORD    number;        /* Number when stored in file */
    WORD    flags;         /* Used to indicate usage when storing */
    WORD    spec;          /* Regular, Tensor, etc. See @ref FunSpecs. */
    WORD    symmetric;     /* > 0 if symmetric properties */
    WORD    node;          /* Location in namenode of #NAMETREE */
    WORD    namesize;      /* Length of the name */
    WORD    dimension;     /* For dimensionality checks */
    WORD    maxnumargs;
    WORD    minnumargs;
    PADPOINTER(2,0,11,0);
} *FUNCTIONS;
```

For this section we are interested in the TABLES tabl element. This is a pointer to a structs that contains the information about the table. This is not a very small struct because it has to store a lot of information:

```
/**
 *
 *  buffers, mm, flags, and prototype are always dynamically allocated,
 *  tablepointers only if needed (=0 if unallocated),
 *  boomlijst and argtail only for sparse tables.
 *
 *  Allocation is done for both the normal and the stub instance (spare),
 *  except for prototype and argtail which share memory.
 */

typedef struct TaBlEs {
    WORD    *tablepointers; /* [D] Start in tablepointers table. */
#ifdef WITHPTHREADS
    WORD    **prototype;    /* [D] The wildcard prototyping for arguments */
    WORD    **pattern;      /* The pattern with which to match the arguments */
#else
    WORD    *prototype;     /* [D] The wildcard prototyping for arguments */
    WORD    *pattern;       /* The pattern with which to match the arguments */
#endif
```

```
MINMAX  *mm;              /* [D] Array bounds, dimension by dimension. # elements = numin
WORD    *flags;           /* [D] Is element in use ? etc. # elements = numind. */
COMPTREE *boomlijst;      /* [D] Tree for searching in sparse tables */
UBYTE   *argtail;         /* [D] The arguments in characters. Starts for tablebase
                             with parenthesis to indicate tail */
struct TaBlEs *spare;     /* [D] For tablebase. Alternatingly stubs and real */
WORD    *buffers;         /* [D] When we use more than one compiler buffer. */
LONG    totind;           /* Total number requested */
LONG    reserved;         /* Total reservation in tablepointers for sparse */
LONG    defined;          /* Number of table elements that are defined */
LONG    mdefined;         /* Same as defined but after .global */
int     prototypeSize;    /* Size of allocated memory for prototype in bytes. */
int     numind;           /* Number of array indices */
int     bounds;           /* Array bounds check on/off. */
int     strict;           /* >0: all must be defined. <0: undefined not substitute */
int     sparse;           /* > 0 --> sparse table */
int     numtree;          /* For the tree for sparse tables */
int     rootnum;          /* For the tree for sparse tables */
int     MaxTreeSize;      /* For the tree for sparse tables */
WORD    bufnum;           /* Each table potentially its own buffer */
WORD    bufferssize;      /* When we use more than one compiler buffer */
```

```
    WORD    buffersfill;    /* When we use more than one compiler buffer */
    WORD    tablenum;       /* For testing of tableuse */
    WORD    mode;           /* 0: normal, 1: stub */
    WORD    numdummies;     /*  */
    PADPOINTER(4,8,6,0);
} *TABLES;
```

The (N)Table statement is read in the routine DoTable in the file names.c. Most of the variables are explained there in the commentary. We will only pay attention to a few: the most important ones. To begin with we have several types of tables: regular tables and sparse tables and each can again be a normal table or defined by a tablebase. And in addition tables can have function arguments, potentially with wildcards. Let us start with a regular table. It is defined as in

```
    Table tab(1:10,-4:5,1:3);
```

The boundaries for each dimension are stored in a struct of type MINMAX which is defined by

```
typedef struct MiNmAx {
    WORD mini;          /* Minimum value */
    WORD maxi;          /* Maximum value */
    WORD size;          /* Value of one unit in this position. */
} MINMAX;
```

in which size indicates the ‘value’ of each element if we construct a one dimensional array to store the elements as in

```
    tab(i1,i2,i3) -> tablepointers[i1*mm[0]+i2*mm[1]+i3*mm[2]]
```

which is computed by

```
    for ( i = T->numind-1, x = 1; i >= 0; i-- ) {
        T->mm[i].size = x;  /* Defines increment in this dimension */
        x *= T->mm[i].maxi - T->mm[i].mini + 1;
    }
```

Note that the fact that x is a WORD size integer limits the size of a table.

For sparse tables we need different parameters. To begin with, we cannot store them in the way we do with specified ranges. Of course we do need the number of table indices. But in more recent versions even that has become more complicated because we can have sparse tables of which the dimension is only limited to a maximum. We store the number of dimensions in numind and add one to it. This will be used for reserving space for each table element and the extra number will point to the table element in cbuf[T->bufnum].rhs, the location where the contents of each fill statement is stored. In the case of when a maximum dimension is specified, we add one to it and flip the sign of the number in numind. This number is to indicate how many indices there are for the table element. Hence we have

```
    Fixed dim:      i1,...,in, numelement     numind =  n+1
    Max dim:      n,i1,...,in, numelement     numind = -n-2
```

Sparse table elements are looked up in a balanced tree. Hence we need this tree which sits in boomlijst.

When a table element is substituted the rhs is inserted in the term as a SUBEXPRESSION subterm. If the table has a function argument and it contains wildcards, these have to be included into this subterm. Hence we have to store the complete SUBEXPRESSION subterm in the TABLES struct together with the lhs pattern that needs to match. They are allocated together. Because during execution the pattern matcher may write in the prototype to indicate the type of match, each worker in TFORM needs its own copy and hence we need to allocate a whole array of them and the total size of the allocation is stored in prototypeSize. The variable argtail is a copy of the arguments following the indices in terms of the character input. This is mainly needed for sparse tables when we want to construct a corresponding tablebase.

For tablebases there need to be extra provisions. The essence of the tablebase is that only the elements that are actually needed will be compiled, because otherwise a few Gbytes of tablebases will take a considerable amount of time at the start of the program (which may still crash on a typo at a later stage), while maybe only a small fraction of the elements is actually needed and that much later in the program. It is psychologically much easier to bear if it crashes as soon as possible. Hence FORM creates for each element in the table base a very small fictitious table element which we call the stub. It is just the function tbl_, indicated by TABLESTUB and it has only the indices of the element for its arguments. This takes hardly any compilation time. At a later moment the user can decide that the program will actively look which tbl_ elements actually occur and compile only those elements of the tablebase. In the element spare we have the stub elements in such a way that we can change between the real table and the stub table by just exchanging the pointers tablepointers and spare. Which one we are dealing with is in the element mode.

The table matching happens in the routine TestSub in the file proces.c. This is initiated with

```
    if ( funnum >= FUNCTION && functions[funnum-FUNCTION].tabl ) {
/*
        Test whether the table catches
        Test 1: index arguments and range. i will be the number
            of the element in the table.
*/
        WORD rhsnumber, *oldwork = AT.WorkPointer;
        WORD ii, *p, *pp, *ppstop;
        MINMAX *mm;
        T = functions[funnum-FUNCTION].tabl;
/*
        Because of tables with a variable number of indices
        we need to make a copy of the pattern.
        If we do this in the WorkSpace we get problems with EndNest.
        This is why we use TermMalloc.
        Now Tpattern is a copy that can be modified.
*/
```

First the index field is to be tested and if that matches something in the table, the arguments have to be tested for a match:

```
caughttable:
    #ifdef WITHPTHREADS
        AN.FullProto = T->prototype[AT.identity];
    #else
        AN.FullProto = T->prototype;
    #endif
        AN.WildValue = AN.FullProto + SUBEXPSIZE;
        AN.WildStop = AN.FullProto+AN.FullProto[1];
        ClearWild(BHEAD0);
        AN.RepFunNum = 0;
        AN.RepFunList = AN.EndNest;
        AT.WorkPointer = (WORD *)(((UBYTE *)(AN.EndNest)) + AM.MaxTer/2);
        if ( AT.WorkPointer >= AT.WorkTop ) {
            MLOCK(ErrorMessageLock);
            MesWork();
            MUNLOCK(ErrorMessageLock);
        }
        wilds = 0;
        if ( MatchFunction(BHEAD Tpattern,t1,&wilds) > 0 ) {
```

If there is indeed a match, the table element is replaced by a SUBEXPRESSION subterm (in AN.FullProto) and Generator can take care of the remaining part because the buffers in which the RHS of a table element are sitting are proper compiler buffers. In the case of the zerofill or onefill options action can be taken to replace the element either by zero, or by one. When the relax option is active, not finding the table element means that it remains unmodified and TestSub will have to keep looking, each time it is called. This is of course inefficient. Maybe this can be improved in the future. One way could be to have a flag in its FUNHEAD that indicates that the occurrence has been checked in this module. These flags can be cleared when a new fill statement is encountered. But also a replace_ function could influence this, or a transform statement, and hence it is not entirely straightforward. It would look a bit like:

```
    if ( functions[*t-FUNCTION].tabl ) {
        if ( ( t[2] & TABLECHECKFLAG ) != 0 ) {
            etc.
        }
    }
```

because the flags of the function in `*t` are in `t[2]`. This looks like a good project for people who suspect that their programs can be made significantly faster with such a check.

Some time ago Takahiro asked for the possibility to have multiple patterns for a zero-dimensional table. This would of course need the option to have an array of patterns. Such an option would then also be applicable for tables with a dimension. And the code in TestSub that checkes for matches with a table element could be modified. Probably the most complicated adaptation would be for the table bases. In essence one needs to keep with each element in the table a number for which pattern is to be used. Of course, once this possibility exists, one would want to extend on it and have only substitutions for a subset of the patterns or of the elements. And one should also realize that if several patterns are possible, some may be inclusive and hence the order of the patterns becomes very important. But it is definitely something to be thought about, but debugging your FORM programs may become more difficult this way.

24 Diagrams

Starting in version 5.0 Form is equiped with the diagram generator of Toshiaki Kaneko. This was originally the diagram generator of the Grace system. More recently Kaneko has rewritten it in C++, written a manual for it and made it generally available. To hook it up to Form it was necessary to define a few new data types. invent a decent Form compatible syntax and write connecting code. This turned out to be relatively easy. The result can be found in the files diagrams.c, diawrap.cc, model.c, model.cc grcc.cc and grcc.h, and several other files like structs.h. New data types are Particle, Vertex and Model. In addition there are new functions named diagrams_, topo_, node_, edge_ block_ and onepi_. Here is an example in which the 2,3,4,5,6 loop propagator diagrams in ϕ^3 theory are constructed.

```
    #define MAXLOOPS "6"
    #define LEGS "2"
    Model PHI3;
        Particle phi,1;
        Vertex phi,phi,phi:g;
    EndModel;
    Vector Q,Q1,...,Q7,p,p0,...,p21;
    Symbols x1,x2,n1,...,n14;
    Indices j1,j2,i1,...,i21;
    Set QQ:Q1,...,Q7;
    Set pp:p1,...,p21;
    Set ii:i1,...,i21;
    Set empty:;
    .global
    #do LOOPS = 2,‘MAXLOOPS’
    L  Fprop‘LOOPS’ = diagrams_(PHI3,{phi,phi},empty,QQ,pp,‘LOOPS’,
                   ‘OnePI_’+‘NoTadpoles_’+‘Symmetrize_’+‘TopologiesOnly_’);
    #if ( ‘LOOPS’ < 4 )
    Print +f +s;
    #endif
    .store

Time =       0.00 sec    Generated terms =          2
          Fprop2         Terms in output =          2
                         Bytes used      =        624

   Fprop2 =
       + 1/4*topo_(1)*node_(1,1,-Q1)*node_(2,1,-Q2)*node_(3,g,Q1,-p1,-p2)*
      node_(4,g,Q2,p1,-p3)*node_(5,g,p2,-p4,-p5)*node_(6,g,p3,p4,p5)
       + 1/4*topo_(2)*node_(1,1,-Q1)*node_(2,1,-Q2)*node_(3,g,Q1,-p1,-p2)*
      node_(4,g,Q2,-p3,-p4)*node_(5,g,p1,p3,-p5)*node_(6,g,p2,p4,p5)
      ;
    #enddo

Time =       0.00 sec    Generated terms =          9
          Fprop3         Terms in output =          9
                         Bytes used      =       3688

   Fprop3 =
       + 1/4*topo_(1)*node_(1,1,-Q1)*node_(2,1,-Q2)*node_(3,g,Q1,-p1,-p2)*
      node_(4,g,Q2,p1,-p3)*node_(5,g,p2,-p4,-p5)*node_(6,g,p3,p4,-p6)*node_(7,
      g,p5,-p7,-p8)*node_(8,g,p6,p7,p8)
       + 1/8*topo_(2)*node_(1,1,-Q1)*node_(2,1,-Q2)*node_(3,g,Q1,-p1,-p2)*
      node_(4,g,Q2,p1,-p3)*node_(5,g,p2,-p4,-p5)*node_(6,g,p3,-p6,-p7)*node_(7
      ,g,p4,p5,-p8)*node_(8,g,p6,p7,p8)
       + 1/4*topo_(3)*node_(1,1,-Q1)*node_(2,1,-Q2)*node_(3,g,Q1,-p1,-p2)*
      node_(4,g,Q2,p1,-p3)*node_(5,g,p2,-p4,-p5)*node_(6,g,p3,-p6,-p7)*node_(7
      ,g,p4,p6,-p8)*node_(8,g,p5,p7,p8)
       + 1/8*topo_(4)*node_(1,1,-Q1)*node_(2,1,-Q2)*node_(3,g,Q1,-p1,-p2)*
      node_(4,g,Q2,-p3,-p4)*node_(5,g,p1,p3,-p5)*node_(6,g,p2,p4,-p6)*node_(7,
      g,p5,-p7,-p8)*node_(8,g,p6,p7,p8)
       + 1/2*topo_(5)*node_(1,1,-Q1)*node_(2,1,-Q2)*node_(3,g,Q1,-p1,-p2)*
      node_(4,g,Q2,-p3,-p4)*node_(5,g,p1,p3,-p5)*node_(6,g,p2,p5,-p6)*node_(7,
      g,p4,-p7,-p8)*node_(8,g,p6,p7,p8)
       + 1/2*topo_(6)*node_(1,1,-Q1)*node_(2,1,-Q2)*node_(3,g,Q1,-p1,-p2)*
      node_(4,g,Q2,-p3,-p4)*node_(5,g,p1,p3,-p5)*node_(6,g,p2,-p6,-p7)*node_(7
      ,g,p4,p6,-p8)*node_(8,g,p5,p7,p8)
       + 1/4*topo_(7)*node_(1,1,-Q1)*node_(2,1,-Q2)*node_(3,g,Q1,-p1,-p2)*
      node_(4,g,Q2,-p3,-p4)*node_(5,g,p1,-p5,-p6)*node_(6,g,p2,p5,-p7)*node_(7
      ,g,p3,p6,-p8)*node_(8,g,p4,p7,p8)
       + 1/16*topo_(8)*node_(1,1,-Q1)*node_(2,1,-Q2)*node_(3,g,Q1,-p1,-p2)*
      node_(4,g,Q2,-p3,-p4)*node_(5,g,p1,-p5,-p6)*node_(6,g,p2,-p7,-p8)*node_(
      7,g,p3,p5,p6)*node_(8,g,p4,p7,p8)
       + 1/8*topo_(9)*node_(1,1,-Q1)*node_(2,1,-Q2)*node_(3,g,Q1,-p1,-p2)*
      node_(4,g,Q2,-p3,-p4)*node_(5,g,p1,-p5,-p6)*node_(6,g,p2,-p7,-p8)*node_(
      7,g,p3,p5,p7)*node_(8,g,p4,p6,p8)
      ;


Time =       0.00 sec    Generated terms =         46
          Fprop4         Terms in output =         46
                         Bytes used      =      23568
```

```
Time =         0.08 sec    Generated terms =         322
          Fprop5           Terms in output =         322
                           Bytes used      =      198368

Time =         0.92 sec    Generated terms =        2793
          Fprop6           Terms in output =        2793
                           Bytes used      =     2010976
    .end
```

There are quite a few options which are given as built in preprocessor variables. The models can be prepared in a .h file. One of the advantages is that one can have the generator produce first a list of all topologies, design preparatory work on those and later generate all diagrams and they will have the same topology numbers. This avoids lots of pattern matching for finding the topologies. In the calculations this finding of the topologies was by far the slowest part in the phase of diagram preparation. A definition of QCD could be:

```
Model QCD;
    Particle qua,QUA,-2;
    Particle gho,GHO,-1;
    Particle glu,+3;
    Vertex qua,QUA,glu:g;
    Vertex gho,GHO,glu:g;
    Vertex glu,glu,glu:g;
    Vertex glu,glu,glu,glu:g^2;
EndModel;
```

Following the manual of Kaneko and the example files in model.c, model.cc, diagrams.c and testasp.cc (the Kaneko test file) it should not be very difficult to use more of the features of the very powerful and very versatile generator.

In the implementation we have made a model a type of set of functions and its elements are the particles and vertices. The particles and vertices are special functions. The diagrams are generated by the special function diagrams_. Its syntax is defined in the manual in the chapter on diagram generation. This chapter contains also more examples.

25 Floating point

As I explained in the notations of the terms, the coefficient was initially designed such that it should also support floating point numbers. But in hardly any of the calculations for which Form was used they were needed and hence their implementation was postponed nearly indefinitely. Until they were needed for the construction of expressions to rewrite Multiple Zeta Values (MZVs) in terms of basis elements. If you are not familiar with MZVs and would like to know about them, you may have a look at the paper about the MZV datamine: Broadhurst, Blümlein and Vermaseren, "The Multiple Zeta Value Data Mine", Comput.Phys.Commun. 181 (2010) 582-625, e-Print: 0907.2557 [math-ph]. In the MZV datamine this is done by solving very large numbers of equations, but the number of variables and equations increases exponentially with the weight. Hence there are practical limits. Francis Brown (on the decomposition of Motivic Multiple Zeta Values", arXiv:1102.1310v2 [math.NT]) managed to design a method in which one does not have to solve for all MZVs at a given weight if one needs to express only a limited number of MZVs in terms of a basis. This method uses almost exclusively rational coefficients, but there is one coefficient that cannot be determined this way. It is the coefficient of the depth 1 basis element. Example (we take the equation from the datamine):

$$\begin{aligned}
\zeta_{8,1,1,5} = {} & +\frac{29056868}{39414375}\zeta_2^6\zeta_3 - \frac{47576}{40425}\zeta_2^5\zeta_5 - \frac{163291}{18375}\zeta_2^4\zeta_7 - \frac{4}{105}\zeta_2^3\zeta_3^3 - \frac{450797}{11025}\zeta_2^3\zeta_9 + \frac{7}{5}\zeta_2^2\zeta_3^2\zeta_5 + \frac{16}{25}\zeta_2^2\zeta_3\zeta_{5,3} \\
& +\frac{454049}{1400}\zeta_2^2\zeta_{11} - \frac{16}{25}\zeta_2^2\zeta_{5,3,3} + 3\zeta_2\zeta_3^2\zeta_7 + \frac{61}{14}\zeta_2\zeta_3\zeta_5^2 + \frac{2}{7}\zeta_2\zeta_3\zeta_{7,3} + \frac{2172853}{420}\zeta_2\zeta_{13} - \frac{2}{7}\zeta_2\zeta_{7,3,3} + \frac{1}{7}\zeta_2\zeta_{5,5,3} \\
& -\frac{33}{4}\zeta_3^2\zeta_9 - \frac{133}{6}\zeta_3\zeta_5\zeta_7 - \frac{25}{9}\zeta_3\zeta_{9,3} - \frac{244}{105}\zeta_5^3 - \frac{359}{105}\zeta_5\zeta_{7,3} + \frac{3}{10}\zeta_7\zeta_{5,3} + \frac{89}{18}\zeta_{9,3,3} + \frac{569}{105}\zeta_{7,3,5} \quad + x\ \zeta_{15}
\end{aligned}$$

The method of Francis Brown lets us verify that the MZV's on the RHS form a basis for the given weight. It also lets us determine all rational coefficients, except for the coefficient of ζ_{15}. This means that if we can evaluate all MZVs involved over the floating point numbers we can determine the missing coefficient as a floating point number and because we know that it has to be a rational, it can be reconstructed provided we have sufficient precision. The program would be:

```
#StartFloat 500,15
L F1 =
 -mzv_(8,1,1,5)
 +29056868/39414375*mzv_(2)^6*mzv_(3)
 -47576/40425*mzv_(2)^5*mzv_(5)
 -163291/18375*mzv_(2)^4*mzv_(7)
 -4/105*mzv_(2)^3*mzv_(3)^3
 -450797/11025*mzv_(2)^3*mzv_(9)
 +7/5*mzv_(2)^2*mzv_(3)^2*mzv_(5)
 +16/25*mzv_(2)^2*mzv_(3)*mzv_(5,3)
 +454049/1400*mzv_(2)^2*mzv_(11)
 -16/25*mzv_(2)^2*mzv_(5,3,3)
 +3*mzv_(2)*mzv_(3)^2*mzv_(7)
 +61/14*mzv_(2)*mzv_(3)*mzv_(5)^2
 +2/7*mzv_(2)*mzv_(3)*mzv_(7,3)
```

```
   +2172853/420*mzv_(2)*mzv_(13)
   -2/7*mzv_(2)*mzv_(7,3,3)
   +1/7*mzv_(2)*mzv_(5,5,3)
   -33/4*mzv_(3)^2*mzv_(9)
   -133/6*mzv_(3)*mzv_(5)*mzv_(7)
   -25/9*mzv_(3)*mzv_(9,3)
   -244/105*mzv_(5)^3
   -359/105*mzv_(5)*mzv_(7,3)
   +3/10*mzv_(7)*mzv_(5,3)
   +89/18*mzv_(9,3,3)
   +569/105*mzv_(7,3,5);
  L F2 = mzv_(15);
  Evaluate mzv_;
  Print +f;
  .sort

 F1 =
    9.12342068779607559001648755754067262393250022224905345406051372588469944\
    91634829703275130822722495241962942249772059922454371995965296661323156O\
    6913925597e+0;
 F2 =
    1.00003058823630702049355172851064506258762794870685817750656993289333322\
    671563422795730723343470175484943669684442492832530297757588781904321794\
    4047700034253;

  Skip F1,F2;
  L X = F1/F2;
  Torational;
  Print +f;
  .end

 X =
    229903169/25200;

0.08 sec out of 0.09 sec
```

The #startfloat initializes the floating point system and allocates arrays for 500 bits of precision. If there is a second number it indicates the maximum weight for MZVs and Euler sums. The functions are only evaluated when the proper command is given. In the second module we divide the numbers and convert the result to a rational. It is a good idea to try this with various precisions to see whether this is stable. With 100 bits the final answer would be

```
   2413649987442716720296838824134 2/26456346795096357002109442 73;
```

while at 150 bits we have already the same answer as with 500 bits. Checking with the datamine we see that the answer `229903169/25200` is indeed correct.

Most of the routines for the floating point operations can be found in the files float.c and evaluate.c. The float.c routines deal with the GMP related things, while the evaluate.c file deals with mpfr related things. The mpfr library contains many functions that can be expanded to arbitrary precision. Hence now FORM has nearly all functions that were originally planned, and names were reserved for. As mentioned before, the size indicator in the terms was originally meant to be even for floating point numbers, but their implementation came after too much time to make this practical. Hence FORM has now a function float_ which stores the floating point number in the format used by the GMP library. This means that there are 4 integer arguments.

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

During normalization the sign is taken away from the second argument and put as the sign of the complete term. This makes writing in sch.c much easier. Wildcarding of functions as in `f?(...)` excludes matches with float_. When the floating point system is turned off with the #endfloat instruction, the float_ function, if present, will survive and become a regular function. Once the #startfloat instruction is issued again, it will resume its function as a floating point number, provided its arguments are legal arguments for such use.

```
 #startfloat 150,15
    .
    .
X =
   229903169/25200;

 #endfloat
 Print +f;
 .end

F1 =
   float_(3,3,1,3104539186058704957034538556033482672955942);

F2 =
   float_(3,3,1,340292775558388953382675519414191995579);

X =
   229903169/25200;
```

The routines in the file float.c are:

```
    #[ 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 :
    ## DoToFloat :
    ## DoFromFloat :
    ## ZeroTable :
    ## ReadFloat :
    ## CheckFloat :
#] Low Level :
#[ Float Routines :
    ## SetFloatPrecision :
    ## PrintFloat :
    ## AddFloats :
    ## MulFloats :
    ## DivFloats :
    ## AddRatToFloat :
    ## MulRatToFloat :
    ## SetupMZVTables :
    ## SetupMPFTables :
    ## ClearMZVTables :
    ## CoToFloat :
```

```
    ## CoToRat :
    ## ToFloat :
    ## ToRat :
#] Float Routines :
#[ Sorting :

    I left out much commentary here. It is in the file.

    ## AddWithFloat :
    ## MergeWithFloat :
#] Sorting :
#[ MZV :

    The functions here allow the arbitrary precision calculation
    of MZV's and Euler sums.
    They are called when the functions mzv_ and/or euler_ are used
    and the evaluate statement is applied.
    The output is in a float_ function.
    The expand statement tries to express the functions in terms of a basis.
    The bases are the 'standard basis for the euler sums and the
    pushdown bases from the datamine, unless otherwise specified.
    ## SimpleDelta :
    ## SimpleDeltaC :
    ## SingleTable :
    ## DoubleTable :
    ## EndTable :
    ## deltaMZV :
    ## deltaEuler :
    ## deltaEulerC :
    ## CalculateMZVhalf :
    ## CalculateMZV :
    ## CalculateEuler :
    ## ExpandMZV :
    ## ExpandEuler :
    ## EvaluateEuler :
#] MZV :
#[ Functions :
    ## CoEvaluate :
    ## GetPi :
    ## GetE :
    ## GetEMconst :
```

```
#] Functions :
```

Conversion from float to rational is the only tricky point. The algorithm is that of repeated fractions, or in other words: we try to write the floating point number x as

```
x = n1 + 1/(n2+1/(n3+1/(n4+....)))
```

The complicated point here is to keep track of how much precision we have remaining while we go deeper and deeper into the expansion. When we suddenly hit on a very big number n_i that is a given fraction of the remaining precision we stop the expansion, set $1/n_i$ equal to zero and work out the remaining fraction. At the moment that the precision of the number we still have is too small we stop anyway.

The basic algorithm for the evaluation of the MZVs and Euler sums is given in the paper by Borwein, Bradley, Broadhurst and Lisonek, Trans.Am.Math.Soc. 353 (2001) 907-941, e-Print: math/9910045 [math.CA]. The essential detail is to evaluate the multiple nested sums as a sequence of single sums of which the results are stored in an array with the upper limit of the sum determining the array element. The sums converge with a factor $\mathcal{O}(1/2)$ in each step. Hence a 1000 bit precision needs an upper limit of $\mathcal{O}(1000)$ for each sum. If one has 6 nested sums that would mean $\mathcal{O}(1000^6)$ steps if the above method would not be used. Now it will be about 6×1000 steps, but one needs a number of intermediate arrays. Additionally one has to be a bit careful with the least significant bits when many numbers have to be added. Hence a few guard bits are added to the precision. Additionally there is a routine that evaluates the constant pi_ and routines are planned for the constants ee_ and em_ (the mathematical $e = 2.71828....$ and the Euler-Mascarponi constant), but I have not yet gotten around programming those. Good algorithms for them can be found in the internet.

26 Potential problems

In this section we will look at a thing that for now should be forbidden. Imagine the following program

```
    Off Statistics;
    Symbols a,b;
    .global
    Global F = a+b;
    .sort
    Symbols c,d;
    Global G = c+d;
    #$t = a+c;
    Print;
    .store

   F =
      b + a;

   G =
      d + c;

    #write "  >>  $t = ‘$t’"
```

```
>>  $t = Z200000027_+a
    Local GG = G;
    .sort
    #write "  >>  $t = ‘$t’"
>>  $t = d+a
    Print;
    .end

   GG =
      d + c;
```

What happened with the variable $t? The variables a and b were declared global, but c and d were not. Hence, after the .store c and d are popped off the variable stack. When we write $t the variable c does not exist and we get some nonsense. It could also have happened that we could have gotten a crash. After this #write operation we read the expression G and this introduces the variables d and c in that order. But that means that what used to be variable c in the first module has become the variable d now. The lesson from this is that if you lift dollar variables over a .store, you better make sure that all its variables have been made global. The alternative would be that if variables are popped off the stack FORM scans all dollar variables during a .store to see whether any of those variables should be kept after all.

27 Things we skipped

We have skipped quite a few features of Form, because the time is only limited and we are talking about a program that was developed over a period of almost 40 years. Hence a few features that could have used attention but did not get any are:

Bracket index Making bracket recovery much faster.

Checkpoints I am not sure they still work properly.

Dictionaries Helps very much in better output, but also in making the program construct code that will be executed at a later stage.

External commands For working with other programs.

If statements How to deal with the conditions.

Special environments Like the term/sort/endterm statements.

Spectators Putting terms out of the way temporarily.

Traces This is maybe the oldest code.

Transform Very useful if one needs to operate many times on the same function.

When I will work out these notes and make them available, I may add a number of these features. There are of course many more features, but most of the remaining ones are conceptually relatively easy once the above notes have been grasped.

28 Exercises part 3

1. Make that

   ```
   L F = ......
   .sort
   L G = F;
   ```

   can be done in parallel. Similar for the spectators.

2. Try to think of what would be needed to define extra table elements during execution. How would this be possible in TFORM or ParFORM?

3. What would be needed to remove dollar variables from the administration? Can you implement that?

4. Change the MesPrint routine in such a way that there is an escape sequence for an ASCII 0 to be added at the end of the format string. Make sure that in a Print "..." statement this passes the compiler and other routines that have to process this.

5. Create a system for printing options for dollar variables as for instance printing one term per line (as in +s) or in raw format as with for terms.

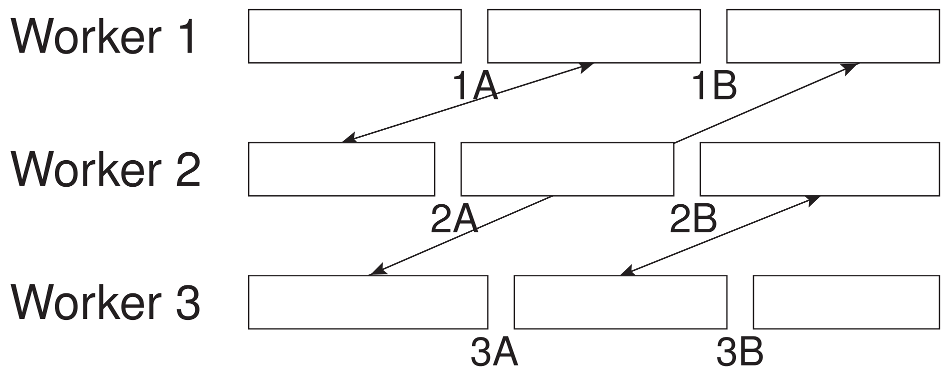
6. Try to make this work:

   ```
     L F = f1(p1+p2)*f1(-p1-p2);
     id f1(p1?)*f1(-p1?) = 1;
   ```

7. When optimizing an expression, it is placed in the workspace. There is currently no decent check on whether there is indeed enough space. Repair this.
8. Make an option that a tablebase is read-only.

28.1 Food for thought

To relieve the bottleneck in TFORM there is a method that should work ”in principle”.



Imagine that we have three workers. The normal thing would be to merge their respective results and writing those to the (common) scratch file. This creates the bottleneck. Assume however that we can determine the points 1A,1B,2A,2B,3A,3B such that the terms in the first pieces of the output of the workers are all smaller than A, and the terms in the last pieces are all greater than B and the pieces in between have only terms greater than A and smaller than B. In that case, if we exchange the offsets to these pieces such that all terms smaller than A become the responsibility of first worker and all terms greater than B become the responsibility of the third worker and all terms in the middle are for the second worker, we can have each of the workers sort their pieces and the final result would be the concatenation of the results of the three workers. They could each write their result to their individual scratch file and use that as their input for the next module. The only work the master would be responsible for would be determining the splitting points, such that each worker gets about the same amount of data.

So why do we not have this method?

The above method is relatively easy to program for fixed size records, specially if they are all different. In that case we would know exactly where each 'term' begins and ends. For us this is not the case. In addition we might even have some data compression making it even more complicated. The solution would be to have a bracket indexing system that keeps track of at least a percentage of the positions of the terms and how to decompress from those positions. It is judged that this method would still need a few more ideas.