Nikhef 2012-004
TTP12-008
SFB/CPP-12-15
March 2012

FORM version 4.0

J. Kuipers[a], T. Ueda[b], J.A.M. Vermaseren[a] and J. Vollinga[a]

[a] *Nikhef Theory Group*
Science Park 105, 1098 XG Amsterdam, The Netherlands

[b] *Institut für Theoretische Teilchenphysik, Karlsruhe Institute of Technology (KIT)*
D-76128 Karlsruhe, Germany

Abstract

We present version 4.0 of the symbolic manipulation system FORM. The most important new features are manipulation of rational polynomials and the factorization of expressions. Many other new functions and commands are also added; some of them are very general, while others are designed for building specific high level packages, such as one for Gröbner bases. New is also the checkpoint facility, that allows for periodic backups during long calculations. Lastly, FORM 4.0 has become available as open source under the GNU General Public License version 3.

1 Introduction

Over the years the symbolic manipulation system FORM [1] has undergone many changes. When its first version was released in 1989, it was capable of dealing with the problems that existed at that time: taking traces of strings of Dirac gamma matrices, executing operations on symbols and dot products of vectors, and manipulating functions with simple arguments. As a successor of Schoonschip [2] FORM went in most of these properties just a bit beyond its predecessor's capabilities. Its main features were already its speed and its facilities for very large expressions.

Over the years the computations done with FORM became more and more complex and hence the program was extended regularly. One of the focuses of development has been the dealing with functions, their argument fields and pattern matching. More communication between the contents of the expressions and the input program has been provided with introducing the $-variables in version 3. This version also contained a complete rewriting of the system that processes the input, which made it much easier to add new commands. In addition the language became a bit more coherent. FORM version 3 was the result of 11 years of experience with different users doing calculations and this had produced a system with many original features. However, one of the capabilities that was missed by many users was the factorization of expressions. This has been added in version 4. Another feature that was requested by many users was the ability to work with rational polynomials as coefficients of terms. In the past this problem was addressed by letting FORM interact with external programs [3] and send such polynomial work to other programs like Fermat [4]. After the external program has finished its work, FORM would receive the results and proceeds from there. In version 4 this is no longer necessary for rational polynomial arithmetic. Nonetheless, the external communications facility still remains and can be used for many other things.

Another feature, that has been added over the years, was running FORM on more than a single processor. This started with the ParFORM [5] project in Karlsruhe. ParFORM is developed to run across different computers over a network using MPI, but it can also be used on computers with multiple cores. For this last type of computers also a multithreaded version of FORM was created, named TFORM [6]. In TFORM most features were easier to implement. The disadvantage of ParFORM is that it has more communication overhead. The disadvantage of TFORM when compared to ParFORM is that TFORM can only run on single multicore computers. When all workers have a heavy interaction with the (usually single) disk, there may be a severe slowdown due to a traffic jam at the disk. When ParFORM runs over a network on several computers, each with their own disk, this problem does not occur. For both ParFORM and TFORM the computing model is such that most programs that are created for regular (sequential) FORM, will not need any modification to benefit from the extra cores by running faster. At times some statements can be added to the program to improve the performance even further.

One of the problems of developing a language is backward compatibility. The transition from version 2 to 3 saw a number of language features changed. To aid the users in the transition a conversion program was provided, but still there were cases in which manual

intervention was necessary. Also, some users had been using, against the instructions of the manual, a variety of bugs in the system that were repaired in version 3. As a result the conversion of old version 2 code was sometimes a bit painful. The introduction of version 4 has none of these problems. All version 3 programs should work with version 4. The only things that have been changed in this respect are some default settings, but they can be restored to their old values in the setup file if wanted.

With the introduction of version 4 some extra facilities become available. FORM is now open source under the GNU General Public License [7] version 3 and there is a publicly accessible CVS repository from which anybody can download the sources[1]. In addition there is quite an amount of documentation. Another facility is the forum[2] on which users and developers can discuss with each other. This has already been very helpful in locating bugs and installation problems.

The outline of this paper is as follows. In section 2 we discuss the generic structure of FORM, TFORM and ParFORM. In section 3 we discuss FORM being open source and address the problem of adding new features. Section 4 is dedicated to the new features that were introduced in the later editions of version 3 and in version 4. In section 5 details of the algorithms used for polynomial algebra are discussed. Some attention is given to the parallel versions in section 6. Section 7 addresses generic facilities and the conclusions are given in section 8.

2 Generic structure

Before going into the new features of version 4 it is best to consider first the internal structure of FORM. The system consists of a number of modules that have a limited amount of interaction with each other, namely

- the preprocessor,
- the compiler,
- the pattern matcher,
- the terms generator,
- the normalization routines, and
- the sorting system.

In addition there are routines that deal with specific tasks such as taking traces of Dirac gamma matrices, initialization, storage, and handling table bases. There are also libraries for many types of standard operations like arithmetic, writing files, and compression. When

[1]The sources are accessible via: `http://www.nikhef.nl/~form/formcvs.php`

[2]The forum is accessible via: `http://www.nikhef.nl/~form/forum`

available, FORM uses libz [8] for the gzip compression routines and the GNU Multiple Precision Arithmetic Library [9] for arithmetic. TFORM uses the POSIX routines for multithreaded processing and ParFORM uses the MPI libraries for network communication.

The preprocessor reads the input and edits this input according to the preprocessor syntax. Each time a statement is completed it is sent to the compiler. Whenever an end-of-module instruction is encountered the combined compiled code in the compiler buffers is executed. Once all active expressions have been processed with this code, the compiler buffer is emptied and the preprocessor continues reading the input.

The compiler is offered complete statements and translates those first into an intermediate code which consists of so-called *tokens*. At this intermediate level some optimizations are performed. Next the tokens are translated into machine code for a virtual machine. The code is placed inside a compiler buffer. During this operation identical subexpressions are recognized to avoid storing them more than once.

The pattern matcher is an extensive piece of code that matches the left hand sides of substitution statements, called *patterns*, with the contents of the terms in the expressions. Its most complicated part is dealing with wildcard variables and backtracking when there are several choices for matching and the choices made so far do not lead to a match.

The center of the program is the terms generator. This is the core of the virtual machine. It controls the branching of the terms when substituting part of a term by one or more terms from either the compiler buffer, from other buffers (like tables, $-variables or other expressions), or from the automatic generation of terms as can happen during the expansion of special functions.

The normalization routines bring terms to normal form. This is not completely trivial, because it also involves action to evaluate special functions.

Finally, the sort routines bring collections of terms into a standard representation so that they form a unique expression. Its special features are that it is in principle disk based and can deal with very large expressions in a relatively short time. Because the representation of the terms is also very compact, FORM can work with very large expressions. For TFORM and ParFORM the sorting system is extended, because with multiple workers involved the sorted results have to be combined into a single expression in the end.

FORM's main objects are expressions. An expression is a sequence of terms stored sequentially on disk unless the expression is small enough to fit inside a cache buffer. The size of this buffer can be set by the user during startup. Each term is self-contained and contains no pointers. During execution no attempt is made to look for common subexpressions in different terms, hence there is no need for reference counts. In principle the terms are processed sequentially. This means that one term is taken from the input expression, the terms generator routine is called and it executes the first statement on it. The first term of the output of this statement is then processed by the generator routine with the second statement and so on. Figure 1 shows this process.

After each statement the current term is normalized to make it ready for the next statement. When there are no more statements, the term is sent to the sorting system and the generator routine goes back one level, picks up the next term on that level, and then executes the next statement on it. This continues until the whole expansion tree has been traversed.

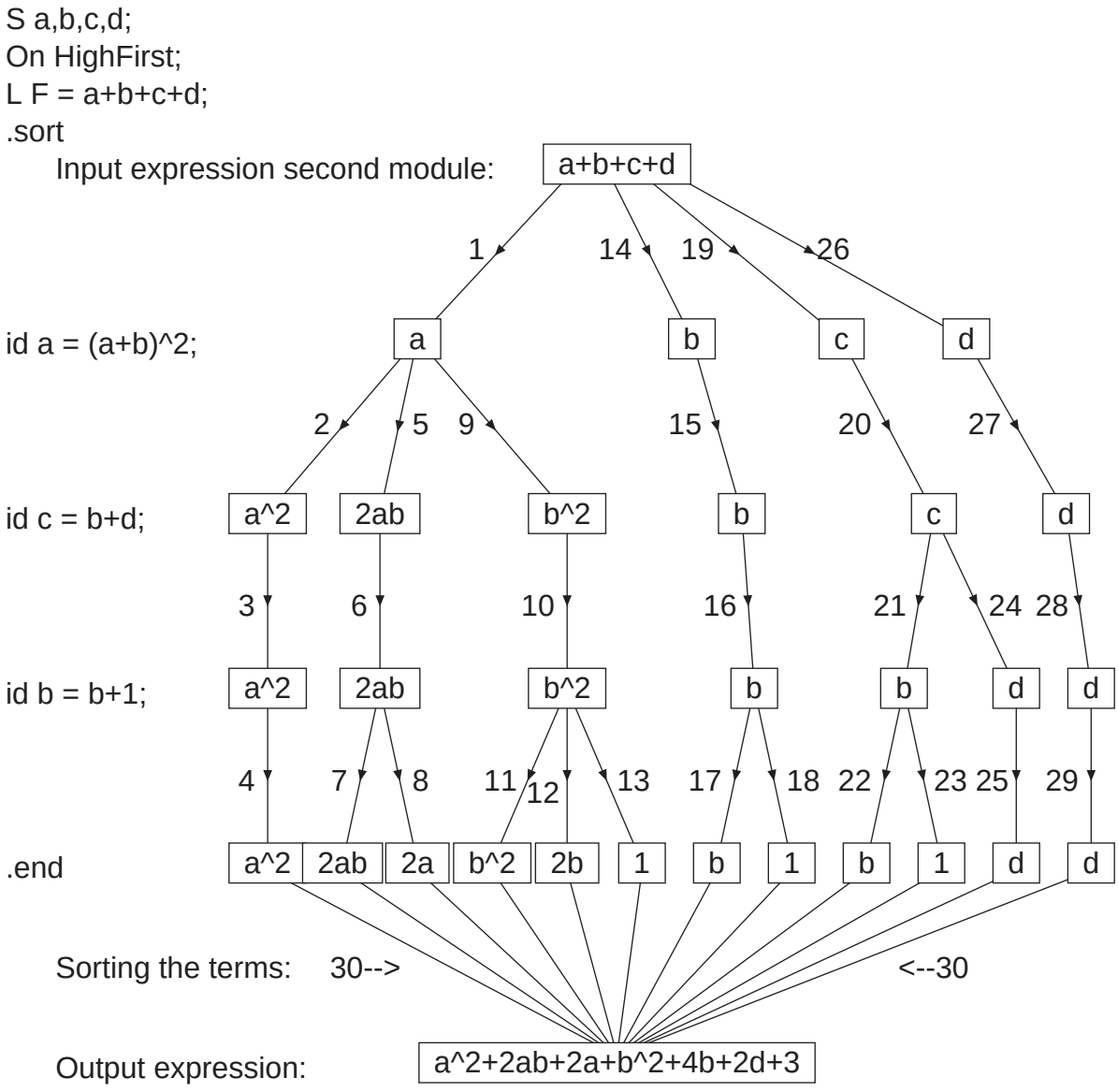

Figure 1: The tree expansion of terms in FORM

The sort routines act semi-independently; they accept terms, put them in buffers, sort the buffers when full, and, if needed, write information to disk. After the last term has been sent to the sorting routines, the sorting is finalized. If the expression is big, this may involve disk-to-disk sorting. As soon as the input is no longer needed, it is destroyed and the output of the current module becomes the input for the next module.

Because all regular operations obtain their input from only a single term they are called local operations. Operations that take input from more than a single term are called non-local. Examples of non-local operations are the compares between terms during the sorting and the factorization of complete expressions. FORM also has some semi-local operations. One of them is the `collect` statement. It places a predefined group of terms inside the argument of a function, which becomes part of a single term. Others involve the manipulation of functions of which one or more arguments contain expressions of more than a single term. When we will look at the polynomial operations we will see that they are inherently non-local although a large class of them can be treated as semi-local. The art of making good FORM programs is often defined by how to construct inherently non-local operations by a combination of local operations and non-local sort instructions. We do not know of any systematic research of this.

3 Open source and adding new features

Starting with version 4, FORM has become an open source program under the GNU General Public License [7] (GPL) version 3. The idea is to allow others to contribute to FORM as well. Therefore it is important to describe shortly how new features can be added to it.

Adding new functions is easiest. There is a header file (`inivar.h`) in which all built-in functions are defined and one needs to add one line there with the name of the function and its internal number which is, given as a name, defined in another header file (`ftypes.h`). Then, depending on the function, its action has to be programmed. This is usually done in the normalization routine. In complicated cases some special routines may have to be added that are called either from the normalization routine or from the generator routine. The source code provides enough examples to illustrate this process.

Adding statements is often relatively simple in FORM. The compiler has to recognize the keyword and send control to the routine that translates the statement. This routine has to be defined and it has to create the proper code for the virtual machine. The generator routine has to recognize this code and give control to the routine that executes the statement. Hence, apart from a few lines of code in other places only two routines have to be constructed.

Adding extra features in the $-variables is similar, but requires more work.

4 New features

During the development of FORM 3 many new features have been added. Most of those have been described before and are counted as belonging to version 3. However, the ones that

were added during the later stages of version 3.3 are actually considered part of version 4 and we will describe them here. For a complete overview of all features one should consult the reference manual.

4.1 Polynomial factorization

In FORM there are three different occurrences of polynomials, namely as arguments of functions, as complete expressions, and as $-variables. These three cases are discussed separately.

4.1.1 Function arguments

When a polynomial is factorized, which is the argument of a function, the factors are returned as a sequence of arguments of the same function. The following program illustrates this.

```
    Symbols a,b,c;
    CFunction f;
    Format NoSpaces;
    Local F = f(3*(a+b)*(a+b+c)*(3*b+2*c));
    Print "<1> %t";
    FactArg f;
    Print "<2> %t";
    .end
<1> +f(6*b*c^2+15*b^2*c+9*b^3+6*a*c^2+21*a*b*c+18*a*b^2+6*a^2*c+9*a^2*b)
<2> +f(3,b+a,c+b+a,2*c+3*b)
```

This use of the `FactArg` statement is different from what it used to be in version 3, when it would only find factors consisting of single terms. For backward compatibility we have provided the possibility to revert to the old behavior with the statement

```
    On OldFactArg;
```

The new `FactArg` behavior is switched on again with

```
    Off OldFactArg;
```

One may also specify the line

```
    OldFactArg ON
```

in the setup file. This would be the easiest way to run old programs.

Dealing with polynomials as function arguments has the drawback that neither the input polynomial nor the combined output polynomials can be larger than the maximum term size.

4.1.2 Expressions

Expressions do not have the size constraint of function arguments, since they are disk based. However, their structure has no natural way to address the factors. To get around this we have selected a way that is in line with features of expressions: the bracket system.

First we adopt the convention that whether an expression is in factorized or unfactorized form is a choice that is made at the output level. The following program illustrates this.

```
Symbols a,b,c;
CFunction f;
Format Nospaces;
Local F = 6*b*c^3+15*b^2*c^2+9*b^3*c+6*a*c^3
         +21*a*b*c^2+18*a*b^2*c+6*a^2*c^2+9*a^2*b*c;
Factorize F;
Print;
.end
```

```
Time =       0.00 sec    Generated terms =          8
               F         Terms in output =          8
                         Bytes used      =        312

Time =       0.00 sec    Generated terms =          9
               F         Terms in output =          9
               factorize Bytes used      =        368

   F=
       (3)
      *(c)
      *(c+b+a)
      *(2*c+3*b)
      *(b+a);
```

The expression is first worked out as shown by the first statistics. Then the result is taken from the output and factorized. The factors are indicated by brackets in terms of the internal symbol `factor_`. They are sorted as indicated by the second statistics. From this point on the expression is kept in factorized form, until it is unfactorized by using the statement `Unfactorize`. The internal representation of this factorized expression is

```
   F=
       factor_*(3)
      *factor_^2*(c)
      *factor_^3*(c+b+a)
      *factor_^4*(2*c+3*b)
      *factor_^5*(b+a);
```

The writing routine does not write the special symbol `factor_`, when the expression is in factorized form, but writes parentheses instead. It should be clear now that the substitution

```
id  a^2 = 1;
```

will have no effect on the factorized expression, since the left hand side of the `id` statement is not present in it.

When a substitution makes a factor zero, it is kept as a zero factor as shown by:

```
Symbols a,b,c;
CFunction f;
Format Nospaces;
Local F = 6*b*c^3+15*b^2*c^2+9*b^3*c+6*a*c^3
         +21*a*b*c^2+18*a*b^2*c+6*a^2*c^2+9*a^2*b*c;
Factorize F;
.sort
id c = -b-a;
Print;
.end
```

```
F=
    (3)
   *(-b-a)
   *(0)
   *(b-2*a)
   *(b+a);
```

Unfactorizing the expression will result in zero, of course.

```
Unfactorize F;
Print;
.end
```

```
F=0;
```

It is also possible to place factorized expressions directly into the input. For this the statements `LocalFactorized` and `GlobalFactorized` are present, which can be abbreviated as follows.

```
Symbols a,b,c;
CFunction f;
LF F = 3*c*(a+b)*(a+b+c)*(3*b+2*c);
Print;
.end
```

```
F =
      ( 3 )
    * ( c )
    * ( b + a )
    * ( c + b + a )
    * ( 2*c + 3*b );
```

Factors are recognized by the multiplication or division signs at ground level, as shown here.

```
Symbols a,b,c;
CFunction f;
LF F = 3/2*(3/2)*c*((a+b)*(a+b+c))*(3*b+2*c);
Print;
.end
```

```
F =
      ( 3 )
    * ( 1/2 )
    * ( 3/2 )
    * ( c )
    * ( b*c + b^2 + a*c + 2*a*b + a^2 )
    * ( 2*c + 3*b );
```

Once we know that the factors are stored as brackets it is easy to refer to them.

```
Local G = F[factor_^3];
Print G;
.end
```

```
G =
   3/2;
```

The number of factors in factorized expression can be obtained by the function `NumFactors_`.

```
Local N = NumFactors_(F);
Print N;
.end
```

```
N =
   6;
```

A value of zero for the number of factors indicates that an expression is in the unfactorized state.

Extra `Bracket` statements will have no effect on factorized expressions. Currently FORM supports only one level of brackets and this level is taken by the factorization. The brackets

of the factorization use automatically the bracket index system to facilitate fast access to the brackets.

If one factorizes an already factorized expression, the system will try to factorize the factors. In addition it combines constant terms and the factors are sorted. If there are factors of zero the result will be zero, unless

```
Factorize(keepzero) F;
```

is used, in which case the zeroes are combined into a single zero as the first factor.

4.1.3 $-variables

The third way of dealing with polynomials is as the content of $-variables. This has yet other requirements, because the $-variables have no access to the bracket system. In the case of the $-variables we also have to take into account that they can be manipulated both at the preprocessor and the execution level. At the same time we have to realize that, although $-variables are not restricted by the maximum term size, they do reside inside memory and therefore cannot be as large as regular expressions.

When a $-variable has been factorized, we keep two copies of it: the unfactorized version and the factorized version. The factors remain in existence until the $-variable is redefined.

The factorization is accomplished with the preprocessor instruction `#FactDollar` or with the statement `FactDollar`. The factors are accessed by giving their number between square brackets after the name of the $-variable as shown underneath. The zeroth element gives the number of factors. One may also use the `NumFactors_` function to obtain the number of factors in a $-variable. Again a zero return value indicates that the variable has not been factorized.

```
    Symbols a,b,c;
    #$X = (a+b+c)*3*(a+b)*(3*b+2*c);
    #FactDollar $X;
    #Message The first  factor of $X is '$X[1]'
~~~The first  factor of $X is 3
    #Message The second factor of $X is '$X[2]'
~~~The second factor of $X is c+b+a
    #Message The third  factor of $X is '$X[3]'
~~~The third  factor of $X is 2*c+3*b
    #Message The fourth factor of $X is '$X[4]'
~~~The fourth factor of $X is b+a
    #Message $X has '$X[0]' factors
~~~$X has 4 factors
    .end
```

One may also use other $-variables or factors of $-variables between the brackets, provided they evaluate into valid factor numbers.

```
Symbols a,b,c,x;
CFunction f;
Local F = f(3*(a+b)*(a+b+c)*(3*b+2*c))
         +f(2*(a+c)*(a+b+c));
id  f(x?$x) = 1;
FactDollar $x;
do $i = 1,$x[0];
  Multiply f($x[$i]);
enddo;
Print +s;
.end

F =
    + f(2)*f(c + a)*f(c + b + a)
    + f(3)*f(b + a)*f(2*c + 3*b)*f(c + b + a)
   ;
```

It should be noted that one cannot use constructions like

```
Multiply <f($x[1])>*...*<f($x[$x[0]])>;
```

because the triple dot operator is a preprocessor facility and `$x[0]` is not defined until the execution phase.

4.2 Rational polynomials

To speed up the manipulation of certain types of expressions FORM is equipped with the `PolyFun` mechanism. This has made programs like Mincer [10] considerably faster. There has been the need to extend this facility to dealing with rational polynomials. Therefore, the statement `PolyRatFun` is introduced in FORM. It is declared in a similar way as the PolyFun:

```
CFunction rat;
PolyRatFun rat;
```

The `PolyFun` and `PolyRatFun` declarations are mutually exclusive. The `PolyRatFun` is considered a special type of `PolyFun` and there can be only one of them at any moment. If one wants to switch back to a mode in which there is neither a `PolyFun` nor a `PolyRatFun` one can use

```
PolyRatFun;
```

to indicate that after this there is no function with that status.

In order for the `PolyRatFun` to be effective, the function needs two arguments and the arguments should contain symbols only. The first argument is then interpreted as the numerator of a fraction and the second as the denominator of that fraction. If there is only

one argument it is seen as the numerator and FORM will add a second argument that has the value one. The use of more than two arguments results in an error.

During the normalization of terms FORM will work out products of more than one `PolyRatFun` as in:

```
Symbols x,y;
CFunction rat;
PolyRatFun rat;
Local F = rat(x,y)*rat(x+1,y+1);
Print;
.end

F =
   rat(x^2 + x,y^2 + y);
```

and during the sorting FORM will add and subtract the fractions when the terms are otherwise identical as in

```
Symbols x,y;
CFunction rat;
PolyRatFun rat;
Local F = rat(x,y)+rat(x+1,y+1);
Print;
.end

F =
   rat(2*x*y + x + y,y^2 + y);
```

The fractions are at all times normalized. This means that the gcd of the numerator and the denominator has been divided out.

Often the denominator of a `PolyRatFun` can be factorized. This should however not be done while the function is still a PolyRatFun, because more than two arguments will cause an error. The following example shows a way of doing it:

```
Symbols x,j,x1,x2;
CFunction rat,num,den;
PolyRatFun rat;
Format NoSpaces;
Local F = sum_(j,1,10,rat(1,x+j));
Print;
.sort

F=
   rat(10*x^9+495*x^8+10560*x^7+127050*x^6+946638*x^5+4510275*x^4+
   13667720*x^3+25228500*x^2+25507152*x+10628640,x^10+55*x^9+1320*
```

```
   x^8+18150*x^7+157773*x^6+902055*x^5+3416930*x^4+8409500*x^3+
   12753576*x^2+10628640*x+3628800);

PolyRatFun;
id rat(x1?,x2?) = num(x1)*den(x2);
FactArg num den;
ChainOut,num;
ChainOut,den;
id num(x1?number_) = x1;
id num(x1?symbol_) = x1;
id den(x1?number_) = 1/x1;
id den(x1?symbol_) = 1/x1;
Print;
.end

F=
   num(11+2*x)*num(966240+2143152*x+1903836*x^2+896368*x^3+247049*
   x^4+41140*x^5+4070*x^6+220*x^7+5*x^8)*den(1+x)*den(2+x)*den(3+x)*
   den(4+x)*den(5+x)*den(6+x)*den(7+x)*den(8+x)*den(9+x)*den(10+x);
```

4.3 ToPolynomial, FromPolynomial and ExtraSymbols

Sometimes it is better to have only symbols in an expression. This is for instance the case when an expression is put in the output for further processing in a C or Fortran program. The `ToPolynomial` statement converts all objects that are not symbols with positive powers into newly defined symbols and puts their definition in memory. The definitions of these symbols can also be printed. These symbols are called *extra symbols* and will have names that do not interfere with any user defined names. Alternatively the user has control over their names and can influence whether they are printed as individual or as array elements. This last option can be rather handy in strong typed languages as C. An example of these statements is:

```
Symbols x,y,z;
CFunction f1,f2;
ExtraSymbols array Ab;
Local F = f1(x+y)*z+f2(z)*(x+y);
ToPolynomial;
.sort
Format C;
#write <example.c> "    int Ab[{‘extrasymbols_’+1}],x,y,z;\n"
#write <example.c> "%X"
#write <example.c> "    F = %e",F
.end
```

This program produces the following output in the file `example.c`:

```
int Ab[3],x,y,z;

Ab[1]=f1(y + x);
Ab[2]=f2(z);

F = z*Ab[1] + y*Ab[2] + x*Ab[2];
```

If the `ExtraSymbols` statement is omitted, the default names for the extra symbols are `Z1_`, `Z2_`, etc. If the `Array` option is omitted or the option `Underscore` is used the notation with the trailing underscore is used in the output, independent of the name selected. The statement `FromPolynomial` undoes the action of the `ToPolynomial` statement.

New features like factorization are implemented internally in terms of symbols only. Therefore the routines that execute the `ToPolynomial` and `FromPolynomial` statements are used also internally for the factorization and in the future they may be used for more features.

4.4 Div_, Rem_, Gcd_, Content_ and Inverse_

Now that FORM has polynomial algebra implemented internally, there is need for several functions, that are available to users, that can do some basic operations. These are:

Div_ The quotient of the division of the first by the second argument. The arguments can be polynomials.

Rem_ The remainder of the division of the first by the second argument. The arguments can be polynomials.

Gcd_ The gcd of all the arguments. The arguments can be polynomials.

Content_ The content of the argument, i.e., the argument divided by the gcd of all terms.

Inverse_ The inverse of the first argument P_1 modulo the second argument P_2, i.e., more explicitly, the polygon P_3 such that $(P_3 \cdot P_1) = 1 \bmod P_2$.

Functions together with their arguments are usually restricted by the condition that they have to fit inside the maximum size of a term. With regular functions and an argument that is an expression, the expression is first expanded and hence subject to this restriction. The first four functions in this category however do not suffer from this limitation, because the expressions are only expanded during the operation itself. The same holds for when the arguments are $-variables.

4.5 Do loops

The `#Do` instruction already exists since the original version of FORM. This is a preprocessor feature that acts purely on the text of the input. Until recently its role at the execution level was taken over by the `Repeat`/`EndRepeat` construction, possibly in combination with `If` statements. With the advent of factorization and the need for organized access to the factors this became too complicated and we have implemented a proper `Do`/`EndDo` construction. The first question is of course what should serve as loop variable. For the original `#Do` instruction we use preprocessor variables, but we do not have these available at execution time. However, we do have the $-variables, which allows us to make the construction:

```
Do $i = 1,5;
   id only x^$i = f(F[factor_^$i]);
EndDo;
```

The boundary parameters in the `Do` statement should be either integers, that can be stored inside one FORM word, or $-variables. They can also be $-variables with factor indicators as in

```
Do $i = 1,$e[0];
   Multiply f($e[$i]);
EndDo;
```

4.6 The transform statement

There are several categories of manipulations of functions with either complicated arguments or many simple arguments that take enormous amounts of time when handled by means of the pattern matcher. An example is the following, where arguments of zero are replaced by ones and vice versa.

```
Symbols x;
CFunction f,g;
Local F = f(1,0,1,0,0,1,0,1);
Multiply g;
repeat id g(?a)*f(x?,?b) = g(?a,1-x)*f(?b);
id f*g(?a) = f(?a);
Print;
.end

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

In this example the program has to use the repeat statement to go 8 times through the pattern matcher, plus a ninth time in the last `id` statement. The `Transform` statement is a whole group of operations in which the much simpler matching is done once and the rest is done internally by FORM:

```
Symbols x;
CFunction f,g;
Local F = f(1,0,1,0,0,1,0,1);
Transform f replace(1,last)=(1,0,0,1);
Print;
.end

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

Basically, the `Transform` statement provides a syntax to deal with large groups of arguments. Each transformation consists of a subkey, indicating its type, followed by arguments that are enclosed by parentheses. Currently, there are 10 different subkeys. After that specific information may follow. The proper syntax of the subkeys is in the manual, but below follows a short description of what is available. The first transformations are quite generally applicable, while the latter are custom-made for a project on Multiple Zeta Values [11, 12].

replace Like the `replace_` function, but allows also replacement of numbers.

encode Combines a range of numerical arguments into a single number over a specified base number.

decode The inverse of the encode subkey: a single argument is converted into a range of arguments.

reverse Reverses a range of arguments.

cycle Applies a cyclic permutation to a range of arguments.

permute Permutes arguments according to a given permutation.

tosumnotation There are two ways to characterize harmonic sums and harmonic polylogarithms. In the sum notation the indices are non-zero integers and in the integral notation there are only the numbers -1,0,1. In the conversion from integral to sum notation a zero adds one to the absolute value of the nonzero number to the right of it as in $0, 0, 0, -1 \to -4$.

tointegralnotation The inverse of tosumnotation, see example below.

islyndon Tests whether the indicated range of arguments forms a Lyndon word[3] according to the ordering of the arguments in FORM. A yes and no argument tell what the main term should be multiplied by, when the answer is yes or no respectively.

tolyndon Will permute the given range in a cyclic manner until it is (if possible) a Lyndon word according to the ordering of the arguments in FORM. Also here yes and no arguments should be specified.

[3]One definition of a Lyndon word is the unique minimal cyclic permutation of a number of objects.

It is possible to use more than one subkey in a single transform statement. This improves efficiency. This is illustrated in an example from a program that manipulates Multiple Zeta Values. To start off with, the original version was:

```
    Symbol x,x1,x2;
    CFunction H,H1;
    Local F = H(3,4,2,6,1,1,1,2);
    Print "<1> %t";
    Repeat id H(?a,x?!{0,1},?b) = H(?a,0,x-1,?b);
    Print "<2> %t";
    Multiply H1;
    Repeat id H(x?,?a)*H1(?b) = H(?a)*H1(?b,1-x);
    id  H*H1(?a) = H(?a);
    Print "<3> %t";
    Repeat id H(x1?,x2?,?a) = H(2*x1+x2,?a);
    Print "<4> %t";
    .end
<1>  + H(3,4,2,6,1,1,1,2)
<2>  + H(0,0,1,0,0,0,1,0,1,0,0,0,0,0,1,1,1,1,0,1)
<3>  + H(1,1,0,1,1,1,0,1,0,1,1,1,1,1,0,0,0,0,1,0)
<4>  + H(907202)
```

In term of a single `transform` statement this becomes:

```
    CF  H;
    L   F = H(3,4,2,6,1,1,1,2);
    Transform H tointegralnotation(1,last),
                replace(1,last)=(0,1,1,0),
                encode(1,last):base=2;
    Print;
    .end

   F =
      H(907202);
```

This last version is faster, more readable, easier to program and less prone to errors.

4.7 Random_ and RanPerm_

`Random_` is a random number generator. It works according to an algorithm called the *additive number generator* as described in [13]. The used subscript pair is $(38, 89)$, which gives a longer cycle than the pair used in the reference. The function is called with an integer argument N. This integer has to be greater than one. The return value is a random number in the range $[1, N]$. The generator can be initialized with the preprocessor instruction

`#SetRandom <number>`. If this instruction is never used the first use of the `Random_` function will use a standard initialization. It should be noticed that in TFORM and ParFORM each worker runs its own version of `Random_` and they are initialized differently, even though this is controlled by the `#SetRandom` instruction. One might argue that this makes the program give different results, depending on whether it is run with FORM or with TFORM, but that would be the case anyway, because the way the terms are distributed over the workers is non-deterministic. Hence, even if there were a unique sequence, the individual terms might not get the same number in the sequence.

An additional useful function is the `RanPerm_` function. Its first argument is the name of the output function. It generates a random permutation of the remaining arguments and puts this permutation in the output function.

```
 CFunction f;
 Local F = RanPerm_(f,1,...,6);
 Print;
 .end
F =
   f(3,1,6,2,4,5);
```

These random functions can be helpful in testing and debugging programs among other things.

4.8 FirstTerm_

The `FirstTerm_` function needs a single argument that is the name of either an expression or a $-variable. It returns the first term in that object. This can be useful in a Gaussian elimination scheme or in a program that computes a Gröbner basis. It is related to the `FirstBracket_` function, which returns what is outside the first bracket of an expression that is in bracketed form.

4.9 Prime_, ExtEuclidean_ and MakeRational_

Several types of calculations can be performed far more efficiently when working modulo prime numbers. Typically, results modulo different prime numbers can be combined with the Chinese remainder theorem to construct solutions over the integers or rationals. For this purpose, the functions `Prime_`, `ExtEuclidean_` and `MakeRational_` are created.

The function `Prime_` generates prime numbers, starting from the maximum positive number that fits inside a single FORM word and working its way down. Numbers that are found are stored in a list, so that they do not have to be recomputed. The function `Prime_(n)` gives the n-th element of that list. If it did not exist yet, FORM has to compute it. Notice that this way of dealing with prime numbers allows only for constrained values of n. The minimum value of a prime number that is allowed is determined by the maximum value that a power of a symbol can have in FORM. On a 64-bits computer this allows for $\mathcal{O}(10^8)$ prime

numbers. One warning: the algorithm for prime number is not particularly fast and hence it might take of the order of a few hours to generate and store them all.

In the extended Euclidean algorithm one does not only determine the greatest common divisor g of two numbers n_1 and n_2, but also two numbers x_1 and x_2 such that $g = x_1 n_1 + x_2 n_2$. In the case that n_1 and n_2 are relative prime (i.e., $g = 1$) x_1 and x_2 are called the modular inverses of n_1 and n_2 because $x_1 n_1 = 1 \bmod n_2$ and so is $x_2 n_2 \bmod n_1$. This algorithm is implemented in FORM in the function `ExtEuclidean_`. It is useful for combining results modulo n_1 and modulo n_2 into a result modulo $n_1 n_2$ as the following example shows.

```
#$p1 = Prime_(1);
#$p2 = Prime_(2);
Symbols x1,x2,x3,x4;
Off Statistics;
Local F = 12345678901234567;
.sort
Local G1 = Mod2_(F,$p1);
Local G2 = Mod2_(F,$p2);
.sort
#$c1 = ExtEuclidean_($p1,$p2);
#Inside $c1
    id ExtEuclidean_(x1?,x2?,x3?,x4?) = x2*x4;
#EndInside
#$c2 = ExtEuclidean_($p1,$p2);
#Inside $c2
    id ExtEuclidean_(x1?,x2?,x3?,x4?) = x1*x3;
#EndInside
#$p3 = $p1*$p2;
Modulus PlusMin '$p3';
Local H = G1*$c1+G2*$c2;
Print;
.end

F =
   12345678901234567;
G1 =
    - 229487668;
G2 =
    - 183496428;
H =
   12345678901234567;
```

`ExtEuclidean_` always returns integer numbers, while often calculations result in rational numbers. To obtain fractions after calculating modulo integers, the function `MakeRational_` is available. It takes two arguments, which both are integer. The function is replaced by the

unique fraction of which both elements are less than the square root of the second argument and that, in calculus modulo this second number would give the same result as the first number modulo the second number. Example:

```
    #$m = prime_(1);
    #write <> "The prime number is %$",$m
The prime number is 2147483587
    Local F = MakeRational_(12345678,$m);
    Print;
    .sort

   F =
      9719/38790;

    Modulus '$m';
    Print;
    .end

   F =
      12345678;
```

4.10 Checkpoints

There are various circumstances in which one can loose a large investment in computer time. One would be a power failure after many hours of running and another would be a syntax error near the end of a program. Whereas the second case is one the user can do something about by first testing the program on small examples, the first case is usually not under control. To avoid the potential loss of big computer resource investments, FORM is equipped with a user controlled backup mechanism. At the end of a module the user has the possibility to make a snapshot of the current state. If, for one reason or another, the program fails at a later stage, it can be restarted from the last snapshot. This snapshot is stored in a disk file and contains the contents of all relevant files and all relevant contents of memory locations. It is tied to a given executable file of FORM, TFORM or ParFORM and only meant to continue execution when the failure condition has been resolved. One should not use it as a practical means of calculation backup. One should also not change the number of workers in TFORM or ParFORM. FORM remembers the reading position in open input files. Hence, altering such a file before this position and changing its length may have disastrous consequences. Changing it after the current reading position should give no problems. This would be a way to repair an error that leads to a crash in the final stages of a program.

The checkpoint feature is activated and deactivated with the statements:

```
On Checkpoint [<OPTIONS>];
Off Checkpoint;
```

The statement `On Checkpoint` will cause a snapshot to be made at the end of the module and each module that follows until a statement `Off Checkpoint` is encountered. If the checkpoints are already active and a new `On Checkpoint` statement is issued, the options of the new statement override the corresponding options of the old statement.

An option of the form

```
<NUMBER>[<UNITS>]
```

specifies a time interval and tells FORM to do the writing at module ends only if more time than this given time has elapsed since the last recovery data write. The units are optional and default to seconds. Possible units are: `s` for seconds, `m` for minutes, `h` for hours, and `d` for days.

FORM can run external programs before and after the recovery data writing. The options to specify the filename of such a program are:

```
runbefore="<FILENAME>"
runafter="<FILENAME>"
run="<FILENAME>"
```

The option `run` specifies the filename for both cases simultaneously. `<FILENAME>` needs to be a valid filename referring to a proper executable. Depending on the operating system this can mean the file needs to have the correct access permissions and to lie in the current search path. The programs will only be run if the condition on the time interval has been fulfilled. This can be used to check for enough disk space, and to compress or move the backups if desired.

The return value of the program run before the data writing will be interpreted. If it is a value unequal to zero or if the execution itself returns an error (e.g., if the executable cannot be found), no data will be written at this module's end. Independently, the program specified to be run after the data writing will always be executed.

The `-R` option in the calling of FORM invokes the recovery procedure after a crash. Imagine that the original program was run with the command

```
form -S my.set long.frm
```

and it crashed, then

```
form -S my.set -R long.frm
```

will continue its execution at the point of the last snapshot.

The writing of the recovery data proceeds in several steps. First, all the data inside FORM is written to an intermediate file. Second, existing files like a store or hide file are copied. At this point, the old recovery data basically gets overwritten. Finally, the intermediate data file is renamed into the proper recovery file. If any problem has occurred, it is signaled to the user. If during the copying of the store, hide, or scratch file anything goes wrong, the old recovery state will be overwritten and left in an unusable, undefined state. An improvement of the situation would be to copy all files first to intermediate files and finally rename them,

but it would triple the amount of space needed on the disk. Because these files can be very large, this was considered impractical. The option `runbefore` can also be used to copy the backup first.

4.11 System independent save files

New is also the portability of the save files (files created with the `Save` statement and recovered with the `Load` statement) between computers with a different architecture. Of course the system does not work in 100% of the cases, because on a computer with a 64-bits architecture one may define many more variables than on a computer with a 32-bits architecture. If there are too many variables, the computer with the 32-bits architecture will give a nametable overflow when reading expressions with too many variables. Similarly there are different restrictions on maximum powers of symbols. The implementation of this portability made it necessary to redefine the format of the save files. Hence old save files will not work with version 4. The best way to convert old save files is by printing their contents to regular text output with the old version of FORM and then reading the resulting expression with a new version of FORM that can put it in a new save file.

5 Polynomial algorithms

For the manipulation of polynomials and the calculation of gcds and factorizations, FORM uses a number of well-known algorithms. To store the polynomials, a degree sparse and variable dense representation is used [14]. Degree sparse indicates that only non-zero terms are stored, so that the representation of sparse polynomials is efficient. Variable dense indicates that for each monomial a vector with the exponents of all variables is stored. This is slightly inefficient regarding storage, but typically speeds up hard polynomial computations such as factorization. For some algorithms, dense univariate polynomials are stored as an array of coefficients if that increases the efficiency.

Polynomial addition and subtraction is implemented by merging two polynomials. Multiplication and division is performed by using a binary heap of monomials to find the next term quickly and achieve good performance [15]. All polynomial operations can be performed over the integers, modulo a prime number or modulo powers of primes. The latter are used for intermediate results in gcd computations and factorization. The prime powers are cached in a table for efficiency.

To calculate the gcd of two univariate polynomials, Euclid's algorithm is used [16]. For the multivariate case FORM tries a heuristic algorithm a couple of times [17], which usually results in the answer. Upon failure, Zippel's sparse modular algorithm is used [18]. This algorithm calculates univariate gcds modulo prime numbers and constructs from them the multivariate gcd with polynomial interpolation and the Chinese remainder algorithm.

For the factorization of univariate polynomials modulo prime numbers Berlekamp's algorithm is used [19]. To obtain the factorization over the integers Hensel lifting is employed [20]. Eventual spurious factors are combined by a brute force approach. Multivariate polynomials

are reduced to univariate ones by substituting appropriate integers for all but one variable, so that Berlekamp's algorithm can be used. The multivariate factorization is then constructed with the multivariate generalization of Hensel lifting. To circumvent as much of the multivariate lifting as possible, coefficients are predetermined by equating the factorization and the original polynomial [21]. The leading coefficient problem is solved by Wang's algorithm [21].

A caching mechanism speeds up many realistic calculations in which the program attempts to factorize the same polynomials more than once.

6 Parallel versions

The fact that an expression in FORM is a sequence of self-contained terms suggests that they can be processed in parallel for local operations. FORM has two versions that allow the use of more than a single processor: TFORM [6] and ParFORM [5]. TFORM uses multiple threads in a single multicore computer in which the cores share the memory. ParFORM uses the MPI library to establish communication between different processes, which do not share the memory and may be on different computers, and to distribute the calculation over these processes. In principle both programs are supposed to work in the same way from the viewpoint of the user. In practice it was much easier to implement TFORM, because one does not need to send information, each time some new data is needed. The cases that one needs to send data via the MPI on ParFORM include: `redefine` statements, $-variables, expressions appearing in right hand sides of definition or substitution statements, and the global table for converting the extra symbols.

Both the parallel versions have been described before, but ParFORM has always been rather incomplete in the sense that some facilities were either missing or not working properly. With version 4 this has been fixed. It should now be possible to run all TFORM programs also with ParFORM. Of course also all regular sequential FORM programs should run both with TFORM and ParFORM.

The two programs have different benefits. TFORM needs far less communication and makes good use of several cores, but it suffers from the presence of only a single disk. ParFORM needs much more communication, but different computers usually have their own disk and hence there is less chance of slowdown due to traffic jams at the disk(s). It is up to the user to decide what is best for a given problem.

Both versions will suffer from bottlenecks. Some parts of the structure of FORM cannot be parallelized in principle or are hard to parallelize. An example is the final sorting on the master. Hence there will be not much further improvement in the performance beyond the use of a given, problem dependent, number of processors. This is a problem that is worked on.

7 Facilities

With the upgrade to the new version also the manual has been extended. The complete manual comes in 4 versions:

.tex the original LaTeX sources,

.ps the processed postscript file,

.pdf the processed PDF file, and

.html the online version.

This manual is a reference manual and should not be confused with a tutorial. The old tutorial by A. Heck and collaborators is still a very good introduction to FORM, but it suffers from the fact that it was written for version 2. After version 2 the syntax has changed somewhat and also many new features were added. One of the future projects is to convert this tutorial to the syntax of version 4.

To make the open source aspect of FORM more realistic new documentation has been written. It explains about the inner working of FORM and many of its routines. Most routines have a header that is compatible with the documentation system Doxygen. This should help with finding ones way through the sources.

In addition a system has been programmed in the language Ruby to allow certification of new executables. In this system many examples of FORM programs are stored, together with the crucial parts of their output. Making a certification run will test all these examples and whether they still give the correct answer. It is the intention to add more examples in the future.

To facilitate communication between users or between users and authors a forum has been created. Here people can report on new FORM programs, problems with installation or execution, request new features, etc. We had to protect the forum a little bit, because at a given moment there were large numbers of spam attacks. Users should sign up and answer an easy question, before they are admitted and can make posts. Currently the forum can be found at `http://www.nikhef.nl/~form/forum`.

The FORM program is all original code in the C and C++ languages. It uses a few generally available libraries. These are libgmp, librt, zlib, the POSIX multithread system, the MPI library, and, of course, the standard C and C++ libraries.

The GMP (GNU Multi Precision Arithmetic Library) deals with the arithmetic of very large numbers. Actually FORM uses only the multiplication, the division and the gcd routines (all for integers) from this library. If the GMP library is not present, FORM has its own routines which may be slower, because in the GMP library there are some assembler statements that do not have an equally efficient equivalent in the C language. In addition the GMP library uses better algorithms for extremely large numbers. The zlib is used for data (de)compression in the sort file (only if the user requests this) and for the tablebase facility. FORM can be built without it. The POSIX library is used for TFORM. Without it

one cannot build TFORM. The MPI library is used for ParFORM and without it one cannot build ParFORM.

8 Conclusions

The new version 4 brings a giant leap forward in the capabilities of FORM. The facility that people missed most, factorization, has been added. Many new commands and functions pave the way for future calculations that apply significantly different algorithms from those used in the past. And if these features are not sufficient, users may implement their own additions.

Acknowledgments: We are very grateful to the authors of zlib, the GMP library, the MPI library and the POSIX library to make their programs available in such an easy to use way.

The work of JK, JV and JV is part of the research programs of the "Stichting voor Fundamenteel Onderzoek der Materie (FOM)", which is financially supported by the "Nederlandse organisatie voor Wetenschappelijke Onderzoek (NWO)" and the work of TU is part of the research program of the DFG through SFB/TR 9 "Computational Particle Physics".

We would also like to thank the people who helped us with the debugging of the system. They spent much time in the preparation of good and comprehensible bug reports.

References

[1] J.A.M. Vermaseren, arXiv:math-ph/0010025.

[2] Schoonschip was originally written by M. Veltman. See: H. Strubbe, Comput. Phys. Commun. **8** (1974) 1-30.

[3] M. Tentyukov and J.A.M. Vermaseren, Comput. Phys. Commun. **176** (2007) 385-405.

[4] Fermat, R.H. Lewis, `http://www.bway.net/~lewis/`.

[5] M. Tentyukov, D. Fliegner, M. Frank, A. Onischenko, A. Retey, H.M. Staudenmaier and J.A.M. Vermaseren, arXiv:cs/0407066; M. Tentyukov, H.M. Staudenmaier and J.A.M. Vermaseren, Nucl. Instrum. Meth. A **559** (2006) 224-228.

[6] M. Tentyukov and J.A.M. Vermaseren, Comput. Phys. Commun. **181** (2010) 1419-1427.

[7] GNU General Public License, `http://www.gnu.org/copyleft/gpl.html`.

[8] zlib, J. Gailly and M. Adler, `http://zlib.net/`.

[9] GNU Multiple Precision Arithmetic Library, `http://gmplib.org/`.

[10] Mincer is a program for the evaluation of massless three loop propagator diagrams. The original paper is: S.G. Gorishnii, S.A. Larin, L.R. Surguladze and F.V. Tkachev, Comput. Phys. Commun. **55** (1989) 381-408. A much improved FORM version is included in the FORM distribution.

[11] J. Blumlein, D.J. Broadhurst and J.A.M. Vermaseren, Comput. Phys. Commun. **181** (2010) 582-625.

[12] J. Kuipers and J.A.M. Vermaseren, arXiv:1105.1884 [math-ph].

[13] D. Knuth, *The art of computer programming, volume 2*, Addison-Wesley (1981).

[14] R.E. Zippel, *Effective polynomial computation*, Kluwer (1993).

[15] M. Monagan and R. Pearce, Proc. of CASC 2007 (2007) 295-315.

[16] K.O. Geddes, S.R. Czapor and G. Labahn, *Algorithms for computer algebra*, Kluwer (1992).

[17] B.W. Char, K.O. Geddes and G.H. Gonnet, J. Symbolic Comp. **9** (1989) 31-48.

[18] R.E. Zippel, *Probabilistic algorithms for sparse polynomials*, PhD thesis, MIT (1979).

[19] E.R. Berlekamp, Bell Syst. Tech. J. **46** (1967) 1853-1859.

[20] H. Zassenhaus, J. Number Theory **1** (1969) 291-311.

[21] P.S. Wang, Math. Comp. **32** (1978) 1215-1231.