Introduction to FORM

Jos Vermaseren

Part 5
Miscellaneous topics
Gamma matrices

The next examples concern power series expansions. Substituting one series into another can cost much time if this is not done carefully. In the example below we want to substitute the power series for $e^y - 1$ into the power series for $\ln(1+x)$. The whole should result in x if all goes well. First let us be brutish about it:

```
#define N "7"
Symbol i, x(:‘N’), y(:‘N’);
Local ln = -sum_(i,1,‘N’, sign_(i)/i*y^i);
.sort
On Statistics;
id y = sum_(i,1,‘N’,x^i*invfac_(i));
Print;
.end
```

```
Time =       4.65 sec    Generated terms =        127
               ln        Terms in output =          1
                         Bytes used      =         18

   ln =
      x;
```

We see here that the declarations of x and y have a direct power cutoff. This is not so relevant for y, but it is for x. If we don't do this we obtain:

```
#define N "7"
Symbol i, x, y(:‘N’);
Local ln = -sum_(i,1,‘N’, sign_(i)/i*y^i);
.sort
On Statistics;
id y = sum_(i,1,‘N’,x^i*invfac_(i));
Print;
.end
```

```
Time =       6.63 sec    Generated terms =     960799
               ln         Terms in output =         43
                          Bytes used      =        998
   ln =
      x + 5039/40320*x^8 + ... + 1/5782448787773246668800000000*x^49;
```

This shows that most generated terms are the terms that we throw away. Even when terms are thrown away rather quickly, they still need some time. Hence it is even better to become a bit more careful about what we generate and try to avoid the generation of terms that we don't need.

Next we use the expansion

$$x \;\to\; x(1+\frac{x}{2}(1+\frac{x}{3}(1+\frac{x}{4}(1+\cdots))))$$

in a slow way to take advantage of the elimination of powers that are too high. Try to verify that this is indeed what the next program does.

```
#define N "50"
On Statistics;
Symbol i, x(:‘N’), y(:‘N’);
Local ln = -sum_(i,1,‘N’, sign_(i)/i*x^i);
id x = x*y;
#do i = 2,‘N’+1
  id y = 1 + x*y/‘i’;
#enddo
Print;
.end
```

```
Time =       11.05 sec    Generated terms =    1295970
                ln        Terms in output =          1
                          Bytes used      =         18

   ln = x;
```

Even though the above program is already much faster, it can be faster yet by sorting after each step in the expansion.

```
    #define N "50"
    On Statistics;
    Symbol i, x(:‘N’), y(:‘N’);
    Local X = -sum_(i,1,‘N’, sign_(i)/i*x^i);
    id x = x*y;
    #do i = 2,‘N’+1
      id y = 1 + x*y/‘i’;
      .sort:step ‘i’;
    #enddo
    Print;
    .end
********Lots of statistics suppressed*****Try this yourself
Time =       0.11 sec    Generated terms =          1
                X        Terms in output =          1
                         Bytes used      =         18

   X =
      x;
```

Of course one cannot always use such a nice expansion as the 'telescope' expansion for the exponential. In that case one should just feed the proper coefficients one at a time. This is not much of an extra complication. We just need some extra features of FORM. Let us first look at the bracket statement:

```
 Symbols a,b,c;
 Local F = (a+b+c)^3;
 Bracket b;
 Print;
 .end
F =
   + b * ( 3*c^2 + 6*a*c + 3*a^2 )
   + b^2 * ( 3*c + 3*a )
   + b^3 * ( 1 )
   + c^3 + 3*a*c^2 + 3*a^2*c + a^3;
```

We see that FORM has introduced brackets for which powers of b are taken outside, and all other objects are inside the brackets. One can specify many objects in a bracket statement, provided they are symbols, vectors, functions, tensors or dotproducts. The bracket for which nothing is outside is always printed last and the parentheses are omitted.

There is an option with the brackets for which the outside is printed but the contents are represented only by a mentioning of how many terms there are:

```
Symbols a,b,c;
Local F = (a+b+c)^3;
Bracket b;
Print[];
.end

F =
    + b * (  3 terms )
    + b^2 * (  2 terms )
    + b^3 * (  1 term )
    + 1 * (  4 terms );
```

This becomes rather handy when we just need to be the structure of an output and there are very many terms.

```
Symbols a,x,c;
Local F = (a+x+c)^2+x;
Bracket x;
.sort
Local G = F[x]^2-4*F[1]*F[x^2];
Print G;
.end
```

```
G =
   1 + 4*c + 4*a;
```

In this example we use the contents of the brackets of the expression F. This is done by attaching a pair of braces with the outside of the bracket we are interested in between them.

It is possible to use $-variables between the braces, but in this context we will skip that.

Let us now return to our power series substitutions. First we define two power series expansions:

```
#define N "10"
Symbols i,x(:`N'),y(:`N');
Local F1 = sum_(i,1,`N',x^i/i^2);
Local F2 = sum_(i,1,`N',y^i/i^3*sign_(i));
Print;
.end
```

```
F1 =
   x + 1/4*x^2 + 1/9*x^3 + 1/16*x^4 + 1/25*x^5 + 1/36*x^6
   + 1/49*x^7 + 1/64*x^8 + 1/81*x^9 + 1/100*x^10;

F2 =
    - y + 1/8*y^2 - 1/27*y^3 + 1/64*y^4 - 1/125*y^5 + 1/216*y^6
    - 1/343*y^7 + 1/512*y^8 - 1/729*y^9 + 1/1000*y^10;
```

We want to replace x in the first expansion by F2. How to proceed?

```
Bracket y;
.sort
Hide F2;
id x = x*y;
#do i = 1,`N'
  id x = F2[y^`i']+x*y;
  .sort
#enddo
Print;
.end
```

```
F1 =
   - y + 3/8*y^2 - 91/432*y^3 + 983/6912*y^4 - 138583/1296000*y^5 +
  8058173/93312000*y^6 - 14096251129/192036096000*y^7 + 398497867261/
  6145155072000*y^8 - 36735684652159/622196951040000*y^9
   + 761493486574817/13826598912000000*y^10;
```

We have here the new feature 'Hide' which tells FORM to put F2 in a special place in which it is not operated upon and its special bracket structure is preserved. To make F2 active again one can use the 'Unhide' statement (see manual).

Another very useful feature of FORM are the tables. A table is a very special function with automatic substitution rules. Tables must have table elements and must be at least one dimensional. The elements are indicated by numbers. Tables can also be non-commuting. We will only commuting tables in the examples. Let us look at an example:

```
Symbols a,x,n;
Table t1(0:2);
Fill t1(0) = 1+a;
Fill t1(1) = 2+a^2;
Fill t1(2) = 3+a^3;
*
Local F = x+x^2+x^3;
  Print "<1> %t";
id x^n? = x^n*t1(n-1);
  Print "  <2> %t";
Print;
Bracket x;
.end
```

The table is declared as one dimensional, running from zero to two. We fill the elements with a 'fill' statement. One can use normal formula's in the RHS. Don't use however \$-variables or previously defined expressions!

We included some print statements to see what happens.

```
<1>  + x
  <2>  + x
  <2>  + a*x
<1>  + x^2
  <2>  + 2*x^2
  <2>  + a^2*x^2
<1>  + x^3
  <2>  + 3*x^3
  <2>  + a^3*x^3

   F =
       + x * ( 1 + a )
       + x^2 * ( 2 + a^2 )
       + x^3 * ( 3 + a^3 );
```

As one can see, the table elements of t1 are substituted immediately when they occur.
But what happens when an element is asked for that doesn't exist?

```
Symbols a,x,n;
Table t1(0:2);
Fill t1(0) = 1+a;
Fill t1(1) = 2+a^2;
Fill t1(2) = 3+a^3;
Local F = x+x^2+x^3;
id x^n? = x^n*t1(n);
Print;
Bracket x;
.end
```

```
F =
    + x * ( 2 + a^2 )
    + x^2 * ( 3 + a^3 )
    + x^3 * ( t1(3) );
```

We see that the element number 3 is outside the table and/or has not been defined and hence is left untouched.

It is possible to specify that the program should be more strict and give an error message or a warning when either the table hasn't been filled completely or elements are used that are not inside the boundaries:

```
    Symbols a,x,n;
    Table,check,t1(0:2);
    Fill t1(0) = 1+a;
    Fill t1(1) = 2+a^2;
    Fill t1(2) = 3+a^3;
    Local F = x+x^2+x^3;
    id x^n? = x^n*t1(n);
    Print;
    Bracket x;
    .end
Table boundary check. Argument 1
      t1(3)
```

The option strict tells FORM to give an error message when there are undefined elements.

Tables can also have arguments. Just as in regular functions there will only be a match when also the arguments match. These arguments can have wildcards:

```
Symbols a,b,x,y,n,ep;
Table t1(0:2,a?);
Fill t1(0) = 1+a*ep;
Fill t1(1) = 2+a*ep+a^2*ep^2;
Fill t1(2) = 3+2*a*ep+a^2*ep^2+a^3*ep^3;
Local F1 = x+x^2+x^3;
Local F2 = x+x^2+x^3;
if ( expression(F1) ) id x^n? = x^n*t1(n-1,y);
if ( expression(F2) ) id x^n? = x^n*t1(n-1,n);
Print +f;
Bracket x;
.end
```

We see here the wildcard a? in the declaration of t1. In the fill statements we don't have to specify the a?, but we can use the 'a' in the RHS.

Then we use y in the table in F1 and n in F2. Notice that n have a value.

```
F1 =
    + x * ( 1 + y*ep )
    + x^2 * ( 2 + y*ep + y^2*ep^2 )
    + x^3 * ( 3 + 2*y*ep + y^2*ep^2 + y^3*ep^3 );

F2 =
    + x * ( 1 + ep )
    + x^2 * ( 2 + 2*ep + 4*ep^2 )
    + x^3 * ( 3 + 6*ep + 9*ep^2 + 27*ep^3 );
```

There is a second type of tables, called the sparse tables. In their case we just specify the dimension and no space for the elements is reserved in advance. The values however are not restricted to a range (just to the short integers which is either -2^{15} to $2^{15}-1$ on 32 bits systems or -2^{31} to $2^{31}-1$ on 64 bits systems).

One may wonder why we need the first type of tables at all. The sparse tables can handle all types anyway. The difference lies in the speed at which table elements can be identified. In the case of sparse tables FORM has to do a lookup in the list of defined elements. This is not nearly as fast as looking up an array element. And undefined table elements have FORM checking quite frequently.

Finally we will construct an example in which we generate the table elements in a less trivial construction, using \$-variables.

Notice that each time `$dummy` is defined (in the preprocessor) the previous table elements are already known. The RHS of `$dummy` is directly expanded.

```
#define MAX "6"
#define NMAX "10"
Symbols x,n,m,ep(:‘MAX’);
Table,sparse,tp(1,m?);
Fill tp(0) = 1;
#do i = 1, ‘NMAX’
 #$dummy = tp(‘i’-1,m)*(‘i’+ep*m);
 Fill tp(‘i’) = ‘$dummy’;
#enddo
Local F = x^2+x^8;
id  x^n? = x^n*tp(n,1/n);
Bracket x;
Print;
.end
```

```
Time =          0.00 sec    Generated terms =         10
                   F        Terms in output =         10
                            Bytes used      =        154

   F =
       + x^2 * ( 2 + 3/2*ep + 1/4*ep^2 )
       + x^8 * ( 40320 + 13698*ep + 29531/16*ep^2 + 16821/128*ep^3
                + 22449/4096*ep^4 + 567/4096*ep^5 + 273/131072*ep^6 );
```

As one can see, the elements must have been expanded already, or we would have generated much more than the 10 terms that are mentioned.

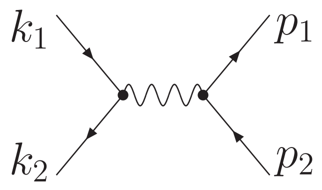
The next topic concerns particle physics specific features. People who are not interested in this can safely skip this section.

For computations in particle physics FORM is equipped with Dirac gamma matrices. These are defined as `g_` . Actually g_ can indicate a whole string of gamma matrices. The first argument is an index (or a short number). After that follow the indices of the string of gamma matrices as in `g_(1,mu1,mu2,mu3,mu4)` $= \gamma^{\mu_1}\gamma^{\mu_2}\gamma^{\mu_3}\gamma^{\mu_4}$. There are special notations:

- g_(n) is the unit matrix in string n.
- g5_(n) is the γ_5 matrix in string n.
- g6_(n) is $1+\gamma_5$ in string n.
- g7_(n) is $1-\gamma_5$ in string n.

To indicate the last three inside a string one can use 5_, 6_ or 7_ as in `g_(1,mu1,7_,mu2,mu3,mu4)`
.

Let us calculate a simple reaction: $e^-e^+ \rightarrow \mu^-\mu^+$ in QED. If we take all particles massless and assume that the $e-$ has momentum k_1, e^+ has k_2, μ^- has p_1 and μ^+ has p_2 we have



```
Vectors k1, k2, p1, p2;
Symbols s, t, u, e;
Indices mu, nu, rho, sigma;
Local  M2 =
*   electron line
         e^2 * g_(1, k2, rho, k1, sigma) *
*   photon propagator
         d_(rho,mu) * d_(sigma,nu) / s^2 *
*   muon spin line
         e^2 * g_(2, p1, mu, p2, nu)
       ;
Trace4,1;
Trace4,2;
Bracket e, s;
Print;
.sort
```

```
M2 =
    + s^-2*e^4 * ( 32*k1.p1*k2.p2 + 32*k1.p2*k2.p1 );

 id  k1.k2 = s/2;
 id  p1.p2 = s/2;
 id  k1.p1 = -t/2;
 id  k2.p2 = -t/2;
 id  k1.p2 = -u/2;
 id  k2.p1 = -u/2;
 Bracket e, s;
 Print;
 .end

M2 =
    + s^-2*e^4 * ( 8*t^2 + 8*u^2 );
```

Of course the interesting command here is the Trace4 command. The Trace4,1; statement causes the taking of the trace of the combination of all gamma matrices that have the spinline 1. In principle gamma matrices of different spinlines commute, so that should not give a problem. Things are a bit more complicated when the spinline is an index. In that case we have not yet fixed the spinline and hence they cannot commute under normal circumstances. However, once we say for instance Trace4,n; it is assumed that all gamma matrices with spinline n commute with all other gamma matrices (or other functions).

The specification Trace4 means that the trace is taken in 4 dimensions. In that case FORM knows some tricks that are specific to 4 dimensions. Like Chisholm identities to pull some traces together:

```
    Vectors k1, k2, p1, p2;
    Indices mu, nu, rho, sigma;
    Local  M2 = g_(1, k2, mu, k1, nu) * g_(2, p1, mu, p2, nu) ;
    Trace4,1;
    Print;
    .end
Time =       0.00 sec    Generated terms =          2
               M2        Terms in output =          2
                         Bytes used      =         50
   M2 = 2*g_(2,p1,k1,nu,k2,p2,nu) + 2*g_(2,p1,k2,nu,k1,p2,nu);
```

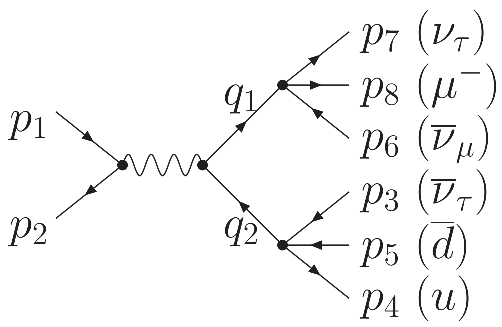
The Chisholm identity that FORM uses is

$$\gamma_\mu Tr[\gamma_\mu S] = 2(S + S^R)$$

in which S is a string of gamma matrices with an odd number of matrices (γ_5 counts for an even number of matrices). S^R is the reversed string. This relation can be used to combine traces with common indices.

The complete trace algorithms are explained in the manual. It also explains about the Tracen statement for traces in n dimensions.

Next we look at the reaction $e^-e^+ \to \tau^-\tau^+ \to u\overline{d}\overline{\nu}_\mu\mu$. We assume that all particles have a mass. The program could look like:



The program could look like:

```
Vectors p1,...,p8,Q,q1,q2;
Indices m1,m2,m3,n1,n2,n3;
Symbol emass, tmass, mass3,...,mass8,s;
On Statistics;
Local F =
*
*   The incoming e- e+ pair. Momenta p1, p2
*
       (g_(1,p2)-emass)*g_(1,m1)*
       (g_(1,p1)+emass)*g_(1,n1)*
*
*   The tau line. tau- is q1, tau+ is q2
*
```

```
      (g_(2,p3)+mass3)*g_(2,m2)*g7_(2)*
      (g_(2,q1)+tmass)*g_(2,m1)*
      (-g_(2,q2)+tmass)*
      g_(2,m3)*g7_(2)*(g_(2,p6)-mass6)*
      g_(2,n3)*g7_(2)*(-g_(2,q2)+tmass)*g_(2,n1)*
      (g_(2,q1)+tmass)*g_(2,n2)*g7_(2)*
*
*   The u d-bar pair. p4 is u, p5 is d-bar
*
      (g_(3,p4)+mass4)*g_(3,m2)*g7_(3)*
      (g_(3,p5)-mass5)*g_(3,n2)*g7_(3)*
*
*   The nu-bar mu pair. p7 is nu_bar, p8 is mu
*
      (g_(4,p7)+mass7)*g_(4,m3)*g7_(4)*
      (g_(4,p8)-mass8)*g_(4,n3)*g7_(4)
```

```
*
*   Finally some normalization
*
       /2^16;
trace4,4;
trace4,3;
trace4,2;
trace4,1;
.sort

Time =       0.01 sec    Generated terms =         164
                F        Terms in output =          27
                         Bytes used      =        1246
    id q1.q1 = tmass^2;
    id q2.q2 = tmass^2;

    id p1.p2 = s/2-emass^2;
    id q1.q2 = s/2-tmass^2;
    print +s;
    .end
```

```
Time =       0.07 sec    Generated terms =          35
                F        Terms in output =          24
                         Bytes used      =         840

   F =
       + 4*p1.p5*p2.p7*p3.p4*p6.p8*tmass^2*s
       - 8*p1.p5*p2.q2*p3.p4*p6.p8*p7.q1*tmass^2
       - 8*p1.p5*p2.q2*p3.p4*p6.p8*p7.q2*tmass^2
       + 4*p1.p7*p2.p5*p3.p4*p6.p8*tmass^2*s
       - 8*p1.p7*p2.q1*p3.p4*p5.q1*p6.p8*tmass^2
       - 8*p1.p7*p2.q1*p3.p4*p5.q2*p6.p8*tmass^2
       - 8*p1.q1*p2.p7*p3.p4*p5.q1*p6.p8*tmass^2
       - 8*p1.q1*p2.p7*p3.p4*p5.q2*p6.p8*tmass^2
       + 8*p1.q1*p2.q2*p3.p4*p5.p7*p6.p8*tmass^2
       + 16*p1.q1*p2.q2*p3.p4*p5.q1*p6.p8*p7.q2
       - 8*p1.q2*p2.p5*p3.p4*p6.p8*p7.q1*tmass^2
       - 8*p1.q2*p2.p5*p3.p4*p6.p8*p7.q2*tmass^2
```

```
 + 8*p1.q2*p2.q1*p3.p4*p5.p7*p6.p8*tmass^2
 + 16*p1.q2*p2.q1*p3.p4*p5.q1*p6.p8*p7.q2
 + 4*p3.p4*p5.p7*p6.p8*emass^2*tmass^2*s
 - 2*p3.p4*p5.p7*p6.p8*tmass^2*s^2
 + 4*p3.p4*p5.p7*p6.p8*tmass^4*s
 - 8*p3.p4*p5.q1*p6.p8*p7.q1*emass^2*tmass^2
 - 8*p3.p4*p5.q1*p6.p8*p7.q2*emass^2*tmass^2
 + 8*p3.p4*p5.q1*p6.p8*p7.q2*emass^2*s
 + 4*p3.p4*p5.q1*p6.p8*p7.q2*tmass^2*s
 - 8*p3.p4*p5.q2*p6.p8*p7.q1*emass^2*tmass^2
 + 4*p3.p4*p5.q2*p6.p8*p7.q1*tmass^2*s
 - 8*p3.p4*p5.q2*p6.p8*p7.q2*emass^2*tmass^2
;
```

Notice that the number of terms generated is rather modest ad that all masses of the decay products drop out. What happens if we don't apply tricks like the Chisholm identity? This is an option in the Trace4 statement:

```
    trace4,nocontract,4;
    trace4,nocontract,3;
    trace4,nocontract,2;
    trace4,nocontract,1;
    .sort
Time =       0.55 sec    Generated terms =     149460
                F        Terms in output =       6891
                         Bytes used      =     231410
    Contract,0;
    .sort

Time =       0.61 sec    Generated terms =      40027
                F        Terms in output =       4223
                         Bytes used      =     150478
    id q1.q1 = tmass^2;
    id q2.q2 = tmass^2;
    id p1.p2 = s/2-emass^2;
    id q1.q2 = s/2-tmass^2;
```

```
if ( count(e_,1) ) Multiply x;
Bracket x;
print[];
.end

Time =       0.63 sec    Generated terms =       5528
                F         Terms in output =       4343
                          Bytes used      =     129832

   F =
       + x * (  1307 terms )

       + 1 * (  3036 terms );
```

We see that there are 1307 terms with a Levi-Civita tensor in it. And lots of terms without. Is this indeed the same answer as we obtained before?

Yes! But it is very difficult to prove it. It involves identities like the Schouten identity that we derived before. The only known systematic way to prove that they are equal is to write both expressions out in vector components. This is very messy.