Compactifying formulas with FORM

J.A.M. Vermaseren

Nikhef

in collaboration with
J. Kuipers (Nikhef, Google) and T. Ueda (Karlsruhe)

- Amuse
- First course
- Main dish
- Visit of the chef
- Desert
- Coffee

Amuse

Imagine the following program:

```
Symbols x,y,z;
Format nospaces;
Local F = 6*y*z^2+3*y^3-3*x*z^2+6*x*y*z-3*x^2*z+6*x^2*y;
Print +f;
.end
```

```
F=6*y*z^2+3*y^3-3*x*z^2+6*x*y*z-3*x^2*z+6*x^2*y;
```

For its numerical evaluation this formula needs 18 multiplications and 5 additions. Can we do better?

```
ExtraSymbols,array,w;
Symbols x,y,z;
Off Statistics;
Format O1,stats=ON;
Format nospaces;
Local F = 6*y*z^2+3*y^3-3*x*z^2+6*x*y*z-3*x^2*z+6*x^2*y;
Print +f;
.end
  w(1)=y*z;
  w(2)=-z+2*y;
  w(2)=x*w(2);
  w(3)=z^2;
  w(1)=w(2)-w(3)+2*w(1);
  w(1)=x*w(1);
  w(2)=y^2;
  w(2)=2*w(3)+w(2);
  w(2)=y*w(2);
  w(1)=w(2)+w(1);
```

```
      F=3*w(1);
*** STATS: original  1P 16M 5A : 23
*** STATS: optimized 0P 10M 5A : 15
```

We see here several options of FORM, but the one to concentrate on is the statement

Format O1,stats=ON;

As can be seen, this gives an output that needs fewer operations for its evaluation. And just like the compiler, FORM has several optimization levels, at increasing cost:

```
ExtraSymbols,array,w;
Symbols x,y,z;
Off Statistics;
Format O2,stats=ON;
Format nospaces;
Local F = 6*y*z^2+3*y^3-3*x*z^2+6*x*y*z-3*x^2*z+6*x^2*y;
Print +f;
.end
  w(1)=z^2;
  w(2)=2*y;
  w(3)=z*w(2);
  w(2)=-z+w(2);
  w(2)=x*w(2);
  w(2)=w(2)-w(1)+w(3);
  w(2)=x*w(2);
  w(3)=y^2;
  w(1)=2*w(1)+w(3);
  w(1)=y*w(1);
```

```
      w(1)=w(1)+w(2);
      F=3*w(1);
*** STATS: original  1P 16M 5A : 23
*** STATS: optimized 0P 9M 5A : 14
```

```
    ExtraSymbols,array,w;
    Symbols x,y,z;
    Off Statistics;
    Format O3,stats=ON;
    Format nospaces;
    Local F = 6*y*z^2+3*y^3-3*x*z^2+6*x*y*z-3*x^2*z+6*x^2*y;
    Print +f;
    .end
      w(1)=x+z;
      w(2)=2*y;
      w(3)=w(2)-x;
      w(1)=z*w(3)*w(1);
      w(3)=y^3;
      w(2)=x^2*w(2);
      w(1)=w(1)+w(3)+w(2);
      F=3*w(1);
*** STATS: original  1P 16M 5A : 23
*** STATS: optimized 1P 6M 4A : 12
```

To most people at this conference it will be immediately clear that such a facility can be very useful, provided it can handle very lengthy expressions as well.

First course

The examples we are going to look at come from two sources.

1. Resolvents or Sylvester determinants. This is rather mathematical, but we use these to compare with a recent article that introduced a new technique in the field of formula simplification (Leiserson at al.)[1].
2. The GRACE system. Here we take formulas that are part of one loop calculations with all masses and several gauge parameters included. In the worst case we have several million terms.

[1] "Efficient Evaluation of Large Polynomials", C.E. Leiserson, L. Li, M.M. Maza, Y. Xie, LNCS **6327** (2010) 342–353.

A Sylvester determinant can be used to determine whether two nonlinear equations in the same variable have a simultaneous solution. When the equations are

$$\begin{aligned} E_1^{(a)} &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 \\ E_2^{(b)} &= b_0 + b_1 x + b_2 x^2 \end{aligned} \tag{1}$$

the matrix looks like

$$\begin{matrix} a_0 & a_1 & a_2 & a_3 & 0 \\ 0 & a_0 & a_1 & a_2 & a_3 \\ b_0 & b_1 & b_2 & 0 & 0 \\ 0 & b_0 & b_1 & b_2 & 0 \\ 0 & 0 & b_0 & b_1 & b_2 \end{matrix}$$

and the resolvent is its determinant. In terms of the a_i and b_j parameters it can be a rather messy formula. In the examples here we try to write such a formula with as few operations as we can manage. Of course, if one is merely interested in obtaining a numerical answer after giving the parameters a value, one can do that much faster by just computing the determinant numerically, but that is not the issue here. In other words: this is a scholastic exercise.

The FORM program looks like (leaving out declarations and some settings):

```
#define N "6"
#define M "5"
.global
*
G  F0 = (a0+<a1*x^1>+...+<a‘N’*x^‘N’>)*(1+<z^1>+...+<z^{‘M’-1}>)
 +y^‘M’*(b0+<b1*x^1>+...+<b‘M’*x^‘M’>)*(1+<z^1>+...+<z^{‘N’-1}>);
id  z = x*y;
Multiply x*y;
B   x,y;
.sort
FillExpression,T=F0(y,x); * Fills the matrix
Drop;
.sort
#call determ(F0,T,{‘N’+‘M’})
.store: Calculate the determinant;
Format O1,stats=on;
L   F1 = F0;
```

```
.sort
#write "Optimization level O1. starttime = ‘time_’"
#Optimize F1;
.store
Format O2,stats=on;
L   F2 = F0;
.sort
#write "Optimization level O2. starttime = ‘time_’"
#Optimize F2;
.store
Format O3,stats=on;
L   F3 = F0;
.sort
#write "Optimization level O3. starttime = ‘time_’"
#Optimize F3;
.store
Format O3,stats=on,mctsnumexpand=5*400,mctsconstant=0.25;
L   F3 = F0;
```

```
.sort
#write "Optimization level O3(5*400). starttime = ‘time_’"
#Optimize F3;
.store
.end
```

The procedure determ calculates the determinant of a two dimensional ‘table’. It is given by

```
#procedure determ(F,T,NN)
 G   'F' = <e_(1)*'T'(1,1)>+...+<e_('NN')*'T'('NN',1)>;
 #if ( {'NN'%4} == 3 )
     Multiply -1;
 #endif
 .sort: determ at step 0;
 #do k = 1,{'NN'-1}
  #if ( {'k'%2} == 0 )
    #redefine kk "{'k'/2+1}"
  #else
    #redefine kk "{'NN'-'k'/2}"
  #endif
  id  e_(i1?,...,i'k'?) =
  #do i = 1,'NN'
    +e_(i1,...,i'k','i')*'T'('i','kk')
  #enddo
    ;
  B   e_;
```

```
    .sort: determ at step 'k';
    Skip; NSkip 'F';
    Keep Brackets;
  #enddo
  id  e_(1,...,'NN') = 1;
#endprocedure
```

It is a variation of a routine that is better described in one of the FORM courses.

The output of the program is

```
    #-
Optimization level O1. starttime = 0.12
*** STATS: original  3920P 40959M 4604A : 53494
*** STATS: optimized 34P 5387M 3509A : 8979
Optimization level O2. starttime = 0.27
*** STATS: original  3920P 40959M 4604A : 53494
*** STATS: optimized 35P 4081M 3222A : 7388
Optimization level O3. starttime = 1.87
*** STATS: original  3920P 40959M 4604A : 53494
*** STATS: optimized 19P 3368M 2834A : 6245
Optimization level O3(5*400). starttime = 73.72
*** STATS: original  3920P 40959M 4604A : 53494
*** STATS: optimized 22P 2714M 2404A : 5167
  207.36 sec out of 207.51 sec
```

The O3 level is statisctical in nature as we will see later in the talk. It can be influenced (adjusted to the problem) by a number of parameters. It is clear that higher levels of optimization cost more time, but give better results.

For various values of the parameters m and n we give here the results.

	7-4 resultant	7-5 resultant	7-6 resultant
Original	29163	142711	587880
FORM O1	4968	20210	71262
FORM O2	3969	16398	55685
FORM O3	3015	11171	36146
Maple	8607	36464	-
Maple tryhard	6451	$\mathcal{O}(27000)$	-
Mathematica	19093	94287	-
Leiserson	4905	19148	65770
Haggies	7540	29125	-

Number of operations after optimization by various programs. The number for the 7-5 resultant with 'Maple tryhard' is taken from Leiserson at al. For the 7-4 resultant they obtain 6707 operations, which must be due to a different way of counting. The same holds for the 7-6 resultant as Leiserson et al. start with 601633 operations. The FORM O3 run used $C_p = 0.07$ and 10×400 tree expansions.

Remark: Probably somebody with much Mathematica experience can do better than the table (without ad hoc programming of course).

As one can see: higher levels of optimization give better results, but also cost more time. And one can also see that the FORM results are better than the syntactic factorization by Leiserson et al. Or any program we could lay our hands on.

Of course, one might argue that also the compiler does optimizations. And that this has been a science for many years. So why bother? Let us have a look at how that works out:

In this table we compare the FORM optimization time with the compilation time and the execution time of the resulting program. Basically this is the only thing that really counts. We study here the 7-6 resolvent.

	Format O0	Format O1	Format O2	Format O3
Operations	587880	71262	55685	36146
FORM time	0.12	1.66	65.43	2398
gcc -O0 time	29.02	6.33	5.64	3.36
run	119.66	13.61	12.24	7.52
gcc -O1 time	3018.6	295.96	199.47	80.82
run	24.30	6.88	6.12	3.58
gcc -O2 time	3104.4	247.60	163.79	65.21
run	21.09	7.00	6.22	3.93
gcc -O3 time	3125.4	276.77	179.24	71.02
run	21.02	6.95	6.19	3.93

FORM run time, compilation times and the time to evaluate the compiled formula 10^5 times (run). All times are in seconds. The O3 option in FORM used $C_p = 0.07$ and 10×400 tree expansions.

As one can see in the table, what is optimal depends very much on how often one would like to evaluate the function. But it should be very clear that FORM outperforms the compiler. And this is after we had to help the compiler a bit, because in the C language there is no decent power function. We defined a simple one that could be inlined for the best result (otherwise the compiler is much slower and the code is also a bit slower).

There is another advantage that should not be underestimated. The optimized compiled code is much shorter. For automatically generated code that can make a big difference in the size of the executable. 2 Gbytes is a pretty bad limit.

The main dish

Now we are going to have a look at how this works out for some typical GRACE formulas.
In the GRACE-Loop system we construct the one loop diagrams and multiply them by the tree graphs. If there are N_L one loop graphs and N_T tree praphs, this gives $N_L \times N_T$ expressions. Each will result in a FORTRAN routine. We will look at just a few individual ones.
Each expression is written in terms of Feynman parameters and the loop momentum is shifted so that the denominator contains just a square of the new loop momentum. Then we bracket in terms of the Feynman parameters and the contents of these brackets we want to compute. These numbers are passed to the loop library which then can produce a number for the diagram as a function of the input parameters.
This means that we need to optimize a large number of expressions (one for each combination of Feynman parameters) and the results are best if this is done simultaneously. FORM can handle that if we bracket in terms of the Feynman parameters.

Hence we have an expression like

$$\begin{aligned} F &= x_1 x_4 (\cdots) \\ &+ x_1 x_4^2 (\cdots) \\ &+ \cdots \\ &+ x_3 Q^2 (\cdots) \\ &+ \cdots \end{aligned}$$

in which we have lengthy formulas inside the brackets. Those formulas must be computed in a Fortran program. A simple example of simultaneous optimization is given in the following program:

```
Symbol x,y,z,h;
ExtraSymbols,array,w;
Off Statistics;
Format O3,Stats=ON;
Local F1 = 2*x^3+3*x^2*y+4*x^2*z+4*x*y*z+6*x*z^2+2*y^2*z+z^3;
Local F2 = 2*x^3+3*x^2*y+3*x^2*z+4*x*y*z+6*x*z^2+2*y^2*z+z^3;
.sort
Local F = h*F1+h^2*F2;
```

```
    .sort
    #optimize F1
*** STATS: original  2P 16M 6A : 26
*** STATS: optimized 0P 10M 6A : 16
    #clearoptimize
    #optimize F2
*** STATS: original  2P 16M 6A : 26
*** STATS: optimized 1P 10M 6A : 18
    #clearoptimize
    Bracket h;
    Print +f;
    .end


      w(1)=4*x + 2*y;
      w(1)=z*w(1);
      w(2)=x^2;
      w(3)=3*w(2);
      w(1)=w(3) + w(1);
```

```
      w(1)=w(1)*y;
      w(4)=x^3;
      w(1)=w(1) + 2*w(4);
      w(4)=z + 6*x;
      w(4)=w(4)*z;
      w(2)=4*w(2) + w(4);
      w(2)=z*w(2);
      w(2)=w(2) + w(1);
      w(3)=w(3) + w(4);
      w(3)=z*w(3);
      w(1)=w(3) + w(1);
      F=h*w(2) + h^2*w(1);
*** STATS: original  4P 32M 12A : 52
*** STATS: optimized 1P 12M 8A : 22
```

We see that when the expressions are simplified individually we need $16+18$ operations, while when they are done together, we need only 22 operations. The brackets in h can be written to the output individually.

For the rest the expressions contain:

- masses
- dotproducts
- Levi-Civita tensors
- coupling constants
- gauge parameters

Of course the FORM optimizer has no idea what the significance of all the variables is. To it formulas are a sophisticated version of chaos.

The examples we use are

1. σ: A typical five-point function diagram from the reaction $e^+e^- \to e^+e^-\gamma$.
2. F_{13}: A not very complicated six-point function diagram from the reaction $e^+e^- \to \mu^+\mu^- u\overline{u}$.
3. F_{24}: A more complicated diagram from the same reaction.
4. 4320×22: The most complicated diagram from the reaction $e^+e^- \to \mu^-\overline{\nu}_\mu u\bar{d}$.

The table shows a number of expressions in increasing size. The number of variables is the sum of the number of Feynman parameters and the number of remaining variables.

	σ	F_{13}	F_{24}	4320×22
Variables	4+11	5+24	5+31	5+118
Expressions	35	56	56	62
Terms	5 717	105 114	836 010	2 427 744
Format O0	33 798	812 645	5 753 030	23 045 841
Format O1	5 615	71 989	391 663	818 973
Format O2	4 599	46 483	233 445	552 582
Format O3	3 380	41 666	195 691	521 305

Results for the physics formulas.

We notice that the bigger the expression, the bigger the improvement factor. Unfortunately the O2 (and O3) algorithms are nonlinear. This means that they take much time. We will see a way to improve on this.

The large number of variables in the 4320×22 expression is caused by the fact that we take each combination of coupling constants as a single variable. The are 90 of those in that expression.

A visit from the chef

So how does FORM do all of the above?
The first step is to implement a Horner scheme as in

$$\begin{aligned} F(x) &= 2+3x+5x^2+7x^3+11x^4+13x^5 \\ &\rightarrow 2+x(3+x(5+x(7+x(11+13x)))) \end{aligned}$$

This reduces the number of multiplications (from 13 to 5).
When there are more variables the Horner scheme is more complicated. We implement it recursively as in

$$\begin{aligned} F(x,y,z) &= f_0(y,z)+f_1(y,z)x+f_2(y,z)x^2+f_3(y,z)x^3+\cdots \\ &\rightarrow f_0(y,z)+x(f_1(y,z)+x(f_2(y,z)+x(f_3(y,z)+\cdots))) \end{aligned}$$

after which we can do the same with the $f_i(y,z)$ in terms of y, and so on. The question is now which order we should choose for the variables.

$$F(x,y,z) = y-3x+5xz+2x^2yz-3x^2y^2z+5x^2y^2z^2$$

Direct evaluation of this polynomial takes 18 multiplications and 5 additions (18,5).

Depending on the order of the Horner scheme we have:

Horner Order	Formula	Operations
x,y,z	$y + x(-3 + 5z + x(y(2z + y(z(-3 + 5z)))))$	8,5
x,z,y	$y + x(-3 + 5z + x(z(y(2 - 3y) + z(5y^2))))$	9,5
y,x,z	$x(-3 + 5z) + y(1 + x^2(2z) + y(x^2(z(-3 + 5z))))$	11,5
y,z,x	$-3x + z(5x) + y(1 + z(2x^2) + y(z(-3x^2 + z(5x^2))))$	14,5
z,x,y	$y - 3x + z(x(5 + x(y(2 - 3y))) + z(x^2(5y^2)))$	11,5
z,y,x	$-3x + y + z(5x + y(2x^2 + y(-3x^2)) + z(y^2(5x^2)))$	14,5

So, how do we know which scheme to use?
If the number of variables is rather limited (when $n!$ is not very large) we can try all orderings. For, say, 30 variables this is obviously not practical. In that case it is most common to use what is called occurrence ordering: the variables are ordered by the number of occurrences in the formula.

When we were programming this we decided to try a number of random orderings, just to see how much better this occurrence ordering was. As an example we used a relatively simple GRACE formula. The result was that occurrence ordering was better than average, but by no means optimal. And sometimes anti-occurrence was better. Hence the big question is:

How to obtain a (near) optimal ordering?

The various multivariate Horner schemes can be represented as a tree in which the first time we have to select one out of n variables, the next time one out of $n-1$ variables, etc. In the end we have a complete ordering with a number that tells us the cost of the evaluation of the formula.

This is very similar to games like for instance Go.

The main difference is of course that in games we have usually an opponent, but if we consider the selection of one variable as two ply, one move followed by a move of the opponent, the simularity should be obvious.

In 2006 there was a breakthrough in game theory w.r.t. how to search through such trees (Kocsis and Szepesvári). It is called Monte Carlo tree search or shortly MCTS. It is based on a weight formula that tells which branch in the tree to select. For equal weights a random number determines which branch to select. The tree is evaluated to the end (also in Go or Chess) and then an evaluation is made. For each branchpoint the average score and the number of visits in the branches are remembered.

The formula for branch i is (UCT stands for Upper Confidence level for Trees):

$$UCT_i = \langle x_i \rangle + 2C_p \sqrt{\frac{2 \log n}{n_i}}$$

The first term in the equation favours trying previously successful branches in the tree. This is called exploitation. The second term favours branches that have not been visited much before (if never, the term is even infinite). This is called exploration. The value of C_p determines the balance between the two.

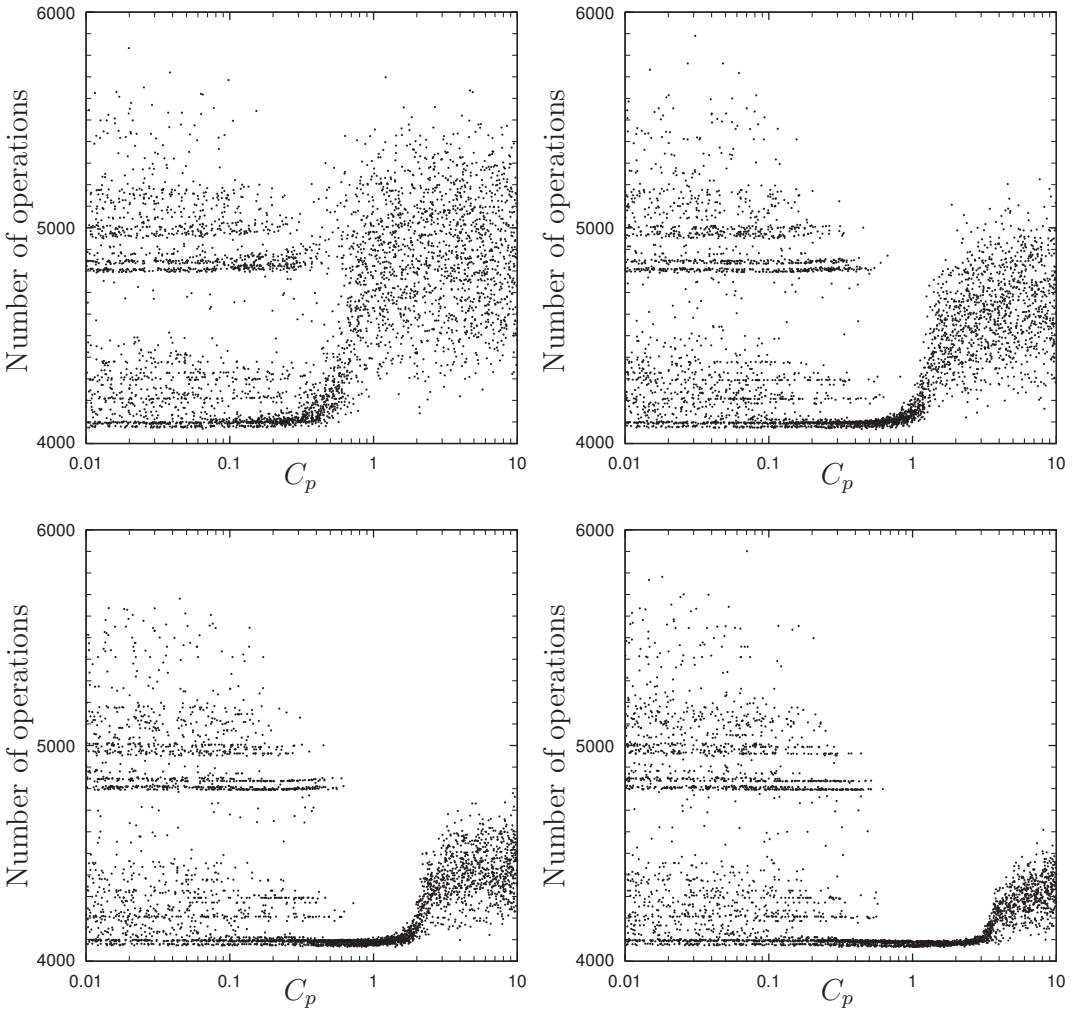
This approach can be successful if positive outcomes are clustered in the tree.

In games this often works because a good move will usually leave many more favourable endpositions than a bad move.

When the value of C_p is too small, we will only sample one seemingly good branch in the tree and eventually end up in a local optimum.

When the value of C_p is too big, we will basically be sampling randomly and forget to pursue branches that seem promising.

The proper value of C_p is problem dependent. It usually requires some experimentation. It would be a separate branch of investigation to see how its best value can be determined automatically.



Scatter plots for 300, 1000, 3000, 10000 points per MCTS run. Each plot has 4000 runs. The formula optimized is one obtained from the σ expression in the previous examples.

The above selection of the Horner scheme would not be very spectacular if we would not apply some more techniques, like common subexpression elimination. Such subexpression eliminations come in varieties of which some take more time than others. Take for instance:

$$\begin{aligned} F(x,y,z) &= y + x(-3 + 5z + x(y(2z + y(z(-3 + 5z))))) \\ &= y + x(Z_1 + x(y(2z + y(zZ_1)))) \\ Z_1 &= -3 + 5z \end{aligned}$$

This insertion saves us one multiplication and one addition.
In FORM we have two types of such insertions:

- Common subexpressions.
- Greedy optimizations.

The second type is far more sophisticated than the first type, but also needs much more CPU type (quadratic algorithm). When FORM works out a Horner scheme it always combines this with the first type. The second type is optional.

Without MCTS the Horner ordering is by 'occurrence' which means that the variables are ordered by the number of occurrences in the formula. Actually the program tries two orderings: occurrence and anti-occurrence.

O1 Occurrence+anti-occurrence followed by common subexpression elimination.

O2 Occurrence+anti-occurrence followed by common subexpression elimination. After that a 'greedy' optimization.

O3 MCTS with common subexpression elimination and greedy optimizations.

With simple expressions that have only a few variables the MCTS can use brute force and try all possibilities. When there are more variables this is of course not possible and it tries a default 1000 times, unless the user specifies a different number.

Desert

The FORM optimization cannot use knowledge that we may have about the formulas. In terms of AI, it does not have domain specific knowledge. Of course, often the best optimizations are based on knowledge about the system, like knowing that in some combination of variables the formula will be shorter.
On the other hand, just introducing lots of new variables makes the work of the MCTS much harder, because the tree becomes bigger. As it turns out, the GRACE output can be greatly improved by making 'shifts' in variables. An example would be that

```
w1 = 2*p1.p2
w2 = me^2
w1+w2 -> w1
```

or: instead of the original w1 we rewrite the formula in terms of the new w1, and very often the number of terms in the expression becomes smaller.
We have written a set of FORM procedures for this and use them before we apply the output optimizations. The results of this are the 'shifted' results in the table.

	σ	F_{13}	F_{24}	4320×22
Variables	4+11	5+24	5+31	5+118
Expressions	35	56	56	62
Terms	5 717	105 114	836 010	2 427 744
Format O0	33 798	812 645	5 753 030	23 045 841
Format O1	5 615	71 989	391 663	818 973
Format O2	4 599	46 483	233 445	552 582
Format O3	3 380	41 666	195 691	521 305
Terms shifted	754	16 439	78 005	193 893
Format O0	4 402+731+15	123 415+605+48	536 127+476+57	1 318 539
Format O1	1 481+409+15	23 459+453+48	68 093+336+57	182 050+1107+582
Format O2	1 146+261+15	17 620+330+48	53 131+229+57	129 740+1107+582
Format O3	1 012+261+15	13 206+322+48	47 379+235+57	119 962+1107+582

Results for the physics formulas in the original and the shifted versions.

Hence, it is best to apply 'domain specific' knowledge first, and after that the FORM optimizations. This is actually also much faster in CPU time, because the shorter formulas after the shifts have an enormous impact on the speed of the O2 and O3 options.

The times for the optimizations are, as we saw before, better than what the compiler could offer us. Yet, the nonlinear optimizations can be rather bad. As an example we take the $4320x22$ diagram (rightmost in the table) which needs 230 sec for the O1 optimization, 77000 seconds for the O2 optimization (going twice through the greedy optimization) and $\mathcal{O}(5\ 10^5)$ sec for O3 in which it does the greedy optimization 10 times at the end. The shift routines take 82000 sec (because there are 90 combinations of coupling constants) after which the O1, O2 and O3 take 18, 680 and 16700 sec respectively (the first two numbers were on a slightly faster computer).

It is far from excluded that the speed of the shift routines can be significantly improved. They are rather experimental.

Coffee

It should be clear that code simplification is not a closed book. Improvements are likely to be found in the future, both in the field of domain specific knowledge and in the improvement of chaotic code simplification. An example:

```
Local F = (a+b)^3+(a+c)^3;
```

Currently the program cannot find such substructures or more complicated varieties of this.

I also expect that now both Maple and Mathematica will improve their performance.

We see also that good optimization can cost significant computer resources. Hence the level of optimization to use depends very much on the number of function evaluations, or the limitations on the size of the executable program.

The application of MCTS in physics should not be restricted to code optimization. It should be considered in any field where one has to search though trees that are far too large for stepping through completely. One such field is the combination of recursion relations in the calculation of all Mellin moments in DIS. Different orderings have completely different properties. They may lead to unacceptably slow programs, or to spurious poles. The use of MCTS for such systems is currently under study.

An interesting spinoff of the implementation of MCTS in the FORM optimization is that the method of MCTS can be studied rather cleanly with it. Much better than in the games of Go or chess. This may lead to improvements of the MCTS method as well, in particular in its parallel applications.

The long paper "Code Optimization in FORM" will appear soon and so will version 4.1 of FORM.