# 1 Creating efficient FORM programs

Creating efficient programs is often a type of art. One starts with defining the problem. Next one takes a direct approach at programming the problem, just to get a feeling of it. If one is lucky, this program solves whatever is needed from it. There are however open ended problems that one would like to solve for as large a value of some parameter as possible. If these problems are in addition exponential in nature, there is a great need of optimizing the program to the highest degree. Hence after the first program one starts with optimizing the code. This is usually done, until one hits some type of limit. Then it becomes time to think about the method. In a series of cycles in which by turn the method and the implementation are further optimized one reaches then a program that will eventually be useful for serious running on big computers.

One remark is important here. Often one wants to compare the performance of several symbolic systems. The results are sometimes rather surprising. One should realize that such compares are only meaningful when an equal amount of expertise and ingenuity is applied for both systems. Hence we do not claim here that the execution times of our examples should be compared with what other people have done with other systems, unless it is clear that those are experts with those systems and used algorithms comparable to what are used here. Maybe they have better algorithms in which case the examples here are a sophisticated exercise in futility. It should always be realized that the quality of the algorithm is dominantly important. Optimization of the code comes in second place.

# 2 A program for generating chromatic polynomials

The problem: We have a lattice of NxN or NxNxN points. The vertices are numbered. We use a standard numbering of the type $i = x + yN$ for two dimensional lattices and $i = x + yN + zN^2$ for cubic lattices. Connections in the rectangular lattice are indicated by e(x1,x2) in which x1 and x2 indicate numbers of vertices. Here we only consider connections for adjacent vertices. We define

$$e(x_1, x_2) \quad = \quad 1 - d(x_1, x_2).$$

When we look at all lines that come together in a given point $x_0$ we have the rule

$$d(x_0, x_1)\ d(x_0, x_2) \cdots d(x_0, x_n) \quad \to \quad d(x_1, x_2)\ d(x_2, x_3) \cdots d(x_{n-1}, x_n)$$

Additional rules are:

$$d(x_1, x_1) \quad = \quad 1$$

and

$$d(x_1, x_2)\ d(x_1, x_2) \quad = \quad d(x_1, x_2)$$

If there is only one line left we have

$$d(x_0, x_1) \quad \to \quad 1$$

and if there is no line left we have

$$1 \quad \to \quad q$$

The variable q will be the variable in the eventual polynomial we are interested in.

The above can be programmed simply in FORM:

```
#-
#define SIZE "6"
CFunctions  d,e,f;
Symbols x,x1,x2,q;
Off Statistics;
*
#do i = 0,`SIZE'^2-1
 L  F`i' = 1
  #if ( {`i'%`SIZE'} != {`SIZE'-1} )
            *e(`i',{`i'+1})
  #endif
  #if ( {(`i'/`SIZE')%`SIZE'} != {`SIZE'-1} )
            *e(`i',{`i'+`SIZE'})
  #endif
        ;
#enddo
id  e(x1?,x2?) = 1-d(x1,x2);
.sort
On Statistics;
Hide;
L   F = 1;
#do i = 0,`SIZE'^2-1
 Multiply F`i';
 id  d(`i',x?) = f(x);
 if ( count(f,1) == 0 );
    Multiply q;
 else;
    repeat id f(x?)*f(x?) = f(x);
    repeat id f(?a)*f(?b) = f(?a,?b);
    repeat id f(x1?,x2?,?a) = d(x1,x2)*f(x2,?a);
    id  d(x1?,x1?) = 1;
    id  f(x?) = 1;
 endif;
 .sort:`i';
#enddo
Print +f +s;
.end
```

We will call this program 1. We assume that there is a setup file with the following contents:

```
MaxTermSize 30K
WorkSpace 5M
LargeSize 100M
SmallSize 10M
ScratchSize 100M
TermsInSmall 1M
LargePatches 1024
FilePatches 1024
```

This program gives execution times as shown in table 1. The space needed is either the amount of bytes used in the output if there is only a single piece of statistics for a given step, or the sum of the bytes used for the last two statistics printed for each step. This was the case for N=7.

We can improve on this program when we realize that the polynomials do not take part in the pattern matching. One way to do this is (program 2):

```
#-
```

| N | Time (sec) | (Disk)space needed |
|---|---|---|
| 4 | 0.15 | 2.5 K |
| 5 | 2.04 | 19 K |
| 6 | 24.7 | 145 K |
| 7 | 261.9 | 2.32 M |

Table 1: Execution times of program 1 on a Pentium 850

```
#define SIZE "6"
CFunctions  d,e,f;
Symbols x,x1,x2,q;
Off Statistics;
*
#do i = 0,‘SIZE’^2-1
 L  F‘i’ = 1
  #if ( {‘i’%‘SIZE’} != {‘SIZE’-1} )
            *e(‘i’,{‘i’+1})
  #endif
  #if ( {(‘i’/‘SIZE’)%‘SIZE’} != {‘SIZE’-1} )
            *e(‘i’,{‘i’+‘SIZE’})
  #endif
        ;
#enddo
id  e(x1?,x2?) = 1-d(x1,x2);
.sort
On Statistics;
Hide;
L   F = 1;
AB  q;
.sort
#do i = 0,‘SIZE’^2-1
 Keep Brackets;
 Multiply F‘i’;
 id  d(‘i’,x?) = f(x);
 if ( count(f,1) == 0 );
    Multiply q;
 else;
    repeat id f(x?)*f(x?) = f(x);
    repeat id f(?a)*f(?b) = f(?a,?b);
    repeat id f(x1?,x2?,?a) = d(x1,x2)*f(x2,?a);
    id  d(x1?,x1?) = 1;
    id  f(x?) = 1;
 endif;
 AB q;
 .sort:‘i’;
#enddo
Print +f +s;
.end
```

and the results are in table 2.

The slight increase in the needed space is due to the bracket information that has to be stored. This is of course more than compensated by the fact that the ‘keep brackets’ statement causes the pattern matching to take place only outside the brackets and not for each individual term.

| N | Time (sec) | (Disk)space needed |
|---|---|---|
| 4 | 0.05 | 2.6 K |
| 5 | 0.48 | 19.6 K |
| 6 | 4.8 | 148 K |
| 7 | 47.26 | 2.33 M |

Table 2: Execution times of program 2 on a Pentium 850

A better way is to use the Polyfun facility. This gives program 3:

```
#-
#define SIZE "6"
CFunctions  d,e,f,acc;
Symbols x,x1,x2,q;
Off Statistics;
*
#do i = 0,`SIZE'^2-1
 L  F`i' = 1
  #if ( {`i'%`SIZE'} != {`SIZE'-1} )
            *e(`i',{`i'+1})
  #endif
  #if ( {(`i'/`SIZE')%`SIZE'} != {`SIZE'-1} )
            *e(`i',{`i'+`SIZE'})
  #endif
        ;
#enddo
id  e(x1?,x2?) = 1-d(x1,x2);
.sort
PolyFun acc;
On Statistics;
Hide;
L   F = 1;
.sort
#do i = 0,`SIZE'^2-1
 Multiply F`i';
 id  d(`i',x?) = f(x);
 if ( count(f,1) == 0 );
    Multiply acc(q);
 else;
    repeat id f(x?)*f(x?) = f(x);
    repeat id f(?a)*f(?b) = f(?a,?b);
    repeat id f(x1?,x2?,?a) = d(x1,x2)*f(x2,?a);
    id  d(x1?,x1?) = 1;
    id  f(x?) = 1;
 endif;
 .sort:`i';
#enddo
PolyFun;
id acc(x?) = x;
Print +f +s;
.end
```

and its execution times are in table 3.

| N | Time (sec) | (Disk)space needed |
|---|---|---|
| 4 | 0.04 | 3.4 K |
| 5 | 0.37 | 24.9 K |
| 6 | 3.24 | 183 K |
| 7 | 26.76 | 1.36 M |
| 8 | 224.6 | 22.8 M |

Table 3: Execution times of program 3 on a Pentium 850

We notice that the use of space for N=7 changed downwards because the sort module of FORM could keep everything inside one buffer and hence there was only a single statistics printed.

The next improvement comes from a new statement in FORM (ChainIn) that manages to avoid one of the repeat statements by an internal loop. We replace the statement

```
repeat id f(?a)*f(?b) = f(?a,?b);
```

by the statement

```
ChainIn f;
```

This gives program 4 with the execution times in table 4.

| N | Time (sec) | (Disk)space needed |
|---|---|---|
| 4 | 0.03 | 3.4 K |
| 5 | 0.33 | 24.9 K |
| 6 | 2.71 | 183 K |
| 7 | 22.82 | 1.36 M |
| 8 | 192.3 | 22.8 M |

Table 4: Execution times of program 4 on a Pentium 850

Next we realize that the number of vertices that are already present in the expression but have not been completed is a measure for the size of the intermediate expressions and hence they should be minimized. This can be done by going through the lattice diagonally. This involves a bit of preprocessor work as can be seen in program 5:

```
#-
#define SIZE "8"
CFunctions  d,e,f,acc;
Symbols x,x1,x2,q;
Off Statistics;
*
#do i = 0,'SIZE'^2-1
 L  F'i' = 1
  #if ( {'i'%'SIZE'} != {'SIZE'-1} )
            *e('i',{'i'+1})
  #endif
  #if ( {('i'/'SIZE')%'SIZE'} != {'SIZE'-1} )
            *e('i',{'i'+'SIZE'})
  #endif
        ;
#enddo
id  e(x1?,x2?) = 1-d(x1,x2);
.sort
```

```
PolyFun acc;
On Statistics;
Hide;
L   F = 1;
.sort
#$num = 1;
#do k = 0,'SIZE'*2-2
#do j = 0,'k'
#if ( ( {'k'-'j'} < 'SIZE' ) && ( 'j' < 'SIZE' ) )
 #redefine i "{'j'*'SIZE'+'k'-'j'}"
 Multiply F'i';
 id  d('i',x?) = f(x);
 id  d(x?,'i') = f(x);
 if ( count(f,1) == 0 );
    Multiply acc(q);
 else;
    repeat id f(x?)*f(x?) = f(x);
    ChainIn f;
    repeat id f(x1?,x2?,?a) = d(x1,x2)*f(x2,?a);
    id  d(x1?,x1?) = 1;
    id  f(x?) = 1;
 endif;
 .sort:'i';
#endif
#enddo
#enddo
PolyFun;
id acc(x?) = x;
Print +f +s;
.end
```

We use the counter \$num to keep an eye on the progress of the program, because now the vertices are not processed sequentially. The results are in table 5.

| N | Time (sec) | (Disk)space needed |
|---|---|---|
| 4 | 0.03 | 2.74 K |
| 5 | 0.26 | 18.1 K |
| 6 | 2.25 | 141 K |
| 7 | 19.66 | 1.15 M |
| 8 | 177.3 | 20.4 M |

Table 5: Execution times of program 5 on a Pentium 850

At this point we seem to be reaching some limit. This usually means that we have to start thinking. The reduction formula is of a rather simple nature and we should be able to prepare its work a bit when we do the previous vertices. From it we can derive that

$$\begin{aligned} d(x_1,x_2)d(x_1,x_3) &= d(x_1,x_2)d(x_2,x_3) \\ d(x_1,x_2)d(x_1,x_2) &= d(x_1,x_2) \\ d(x_1,x_1) &= 1 \end{aligned}$$

When we apply this program 6 will look like:

```
#-
#define SIZE "8"
CFunctions  d,e,f,acc;
Symbols x,x1,x2,x3,q;
Off Statistics;
*
#do i = 0,`SIZE'^2-1
 L  F`i' = 1
  #if ( {`i'%`SIZE'} != {`SIZE'-1} )
            *e(`i',{`i'+1})
  #endif
  #if ( {(`i'/`SIZE')%`SIZE'} != {`SIZE'-1} )
            *e(`i',{`i'+`SIZE'})
  #endif
        ;
#enddo
id  e(x1?,x2?) = 1-d(x1,x2);
.sort
PolyFun acc;
On Statistics;
Hide;
L   F = 1;
.sort
#$num = 1;
#do k = 0,`SIZE'*2-2
#do j = 0,`k'
#if ( ( {`k'-`j'} < `SIZE' ) && ( `j' < `SIZE' ) )
 #redefine i "{`j'*`SIZE'+`k'-`j'}"
 Multiply F`i';
 id  d(`i',x?) = f(x);
 id  d(x?,`i') = f(x);
 if ( count(f,1) == 0 );
    Multiply acc(q);
 else;
    repeat id f(x?)*f(x?) = f(x);
    ChainIn f;
    repeat id f(x1?,x2?,?a) = d(x1,x2)*f(x2,?a);
    id  d(x1?,x1?) = 1;
    repeat;
        id  d(x2?,x1?)*d(x3?,x1?) = d(x2,x3)*d(x3,x1);
        id  d(x1?,x1?) = 1;
    endrepeat;
    repeat;
        id  d(x1?,x2?)*d(x1?,x3?) = d(x2,x3)*d(x1,x2);
        id  d(x1?,x1?) = 1;
    endrepeat;
    id  f(x?) = 1;
 endif;
 .sort:`i',`$num';
 #$num = $num + 1;
#endif
#enddo
#enddo
PolyFun;
```

```
id  acc(x?) = x;
Print +f +s;
.end
```

and its execution times are in table 6.

| N | Time (sec) | (Disk)space needed |
|---|---|---|
| 5 | 0.14 | 8.6 K |
| 6 | 0.68 | 40.1 K |
| 7 | 3.29 | 186 K |
| 8 | 15.5 | 857 K |
| 9 | 75.9 | 8.23 M |

Table 6: Execution times of program 6 on a Pentium 850

There are two more small improvements to be made. Considering that we can do 'work in advance' we can make some simplifications already at the beginning of the program by inserting

at the startup of the program immediately after the replacement of $e(x_1, x_2)$ and it turns out that $q-1$ is a slightly more natural variable which gives coefficients in the polynomials that have fewer digits. This makes that FORM has less work to do. Hence we replace `acc(q)` by `acc([q-1]+1)` and in the end we substitute everything back to q. This gives the final program

```
#-
#define SIZE "8"
CFunctions  d,e,f,acc;
Symbols x,x1,x2,x3,q,[q-1];
Off Statistics;
*
#do i = 0,`SIZE'^2-1
 L  F`i' = 1
  #if ( {`i'%`SIZE'} != {`SIZE'-1} )
            *e(`i',{`i'+1})
  #endif
  #if ( {(`i'/`SIZE')%`SIZE'} != {`SIZE'-1} )
            *e(`i',{`i'+`SIZE'})
  #endif
        ;
#enddo
id  e(x1?,x2?) = 1-d(x1,x2);
repeat;
    id  d(x1?,x2?)*d(x1?,x3?) = d(x2,x3)*d(x1,x2);
    id  d(x1?,x1?) = 1;
endrepeat;
.sort
PolyFun acc;
On Statistics;
Hide;
L   F = 1;
.sort
#$num = 1;
#do k = 0,`SIZE'*2-2
#do j = 0,`k'
```

```
#if ( ( {`k'-`j'} < `SIZE' ) && ( `j' < `SIZE' ) )
 #redefine i "{`j'*`SIZE'+`k'-`j'}"
 Multiply F`i';
 id  d(`i',x?) = f(x);
 id  d(x?,`i') = f(x);
 if ( count(f,1) == 0 );
    Multiply acc([q-1]+1);
 else;
    repeat id f(x?)*f(x?) = f(x);
    ChainIn f;
    repeat id f(x1?,x2?,?a) = d(x1,x2)*f(x2,?a);
    id  d(x1?,x1?) = 1;
    repeat;
        id  d(x2?,x1?)*d(x3?,x1?) = d(x2,x3)*d(x3,x1);
        id  d(x1?,x1?) = 1;
    endrepeat;
    repeat;
        id  d(x1?,x2?)*d(x1?,x3?) = d(x2,x3)*d(x1,x2);
        id  d(x1?,x1?) = 1;
    endrepeat;
    id  f(x?) = 1;
 endif;
 .sort:`i',`$num';
 #$num = $num + 1;
#endif
#enddo
#enddo
PolyFun;
id  acc(x?) = x;
id [q-1] = q-1;
Print +f +s;
.end
```

and its execution times are shown in table 7.

| N | Time (sec) | (Disk)space needed |
|---|---|---|
| 5 | 0.13 | 8.2 K |
| 6 | 0.72 | 38.2 K |
| 7 | 3.19 | 172 K |
| 8 | 15.3 | 802 K |
| 9 | 74.0 | 7.45 M |
| 10 | 355 | 36.0 M |
| 11 | 1705 | 175 M |
| 12 | 8282 | 775 M |

Table 7: Execution times of program 7 on a Pentium 850

Of course we can use this program now also for the three dimensional lattice. In that case the program becomes:

```
#-
#define SIZE "3"
CFunctions  d,e,f,acc;
Symbols x,x1,x2,x3,q,[q-1];
```

```
Off Statistics;
*
#do i = 0,`SIZE'^3-1
 L  F`i' = 1
  #if ( {`i'%`SIZE'} != {`SIZE'-1} )
            *e(`i',{`i'+1})
  #endif
  #if ( {(`i'/`SIZE')%`SIZE'} != {`SIZE'-1} )
            *e(`i',{`i'+`SIZE'})
  #endif
  #if ( {(`i'/{`SIZE'^2})%`SIZE'} != {`SIZE'-1} )
            *e(`i',{`i'+`SIZE'^2})
  #endif
        ;
#enddo
id  e(x1?,x2?) = 1-d(x1,x2);
repeat;
    id  d(x1?,x2?)*d(x1?,x3?) = d(x2,x3)*d(x1,x2);
    id  d(x1?,x1?) = 1;
endrepeat;
.sort
PolyFun acc;
On Statistics;
Hide;
L   F = 1;
.sort
#$num = 1;
#do i1 = 0,`SIZE'*3-3
 #do i2 = 0,`i1'
  #redefine x1 "{`i1'-`i2'}"
  #if ( `x1' < `SIZE' )
   #do i3 = 0,`i2'
    #redefine x2 "{`i2'-`i3'}"
    #if ( ( `x2' < `SIZE' ) && ( `i3' < `SIZE' ) )
     #redefine i "{`x1'+`x2'*`SIZE'+`i3'*`SIZE'^2}"
 Multiply F`i';
 id  d(`i',x?) = f(x);
 id  d(x?,`i') = f(x);
 if ( count(f,1) == 0 );
    Multiply acc([q-1]+1);
 else;
    repeat id f(x?)*f(x?) = f(x);
    ChainIn f;
    repeat id f(x1?,x2?,?a) = d(x1,x2)*f(x2,?a);
    id  d(x1?,x1?) = 1;
    repeat;
        id  d(x2?,x1?)*d(x3?,x1?) = d(x2,x3)*d(x3,x1);
        id  d(x1?,x1?) = 1;
    endrepeat;
    repeat;
        id  d(x1?,x2?)*d(x1?,x3?) = d(x2,x3)*d(x1,x2);
        id  d(x1?,x1?) = 1;
    endrepeat;
    id  f(x?) = 1;
```

```
 endif;
 .sort:'i','$num';
 #$num = $num + 1;
    #endif
   #enddo
  #endif
 #enddo
#enddo
PolyFun;
id  acc(x?) = x;
id [q-1] = q-1;
Print +f +s;
.end
```

Its execution times are

| N | Time (sec) | (Disk)space needed | Computer | FORM version |
|---|---|---|---|---|
| 2 | 0.01 | 668 | P850 | 3.1 |
| 3 | 15.4 | 503 K | P850 | 3.1 |
| 4 | 261654 | 20.5 G | P1700 | 3.1 |
| 4 | 283407 | 11.4 G | P1700 | 3.2 (gzip 6) |

Table 8: Execution times of program 8

The next improvement is valid only when we do not work at finite temperature. In that case we have

$$e(x_1, x_2)d(x_1, x_2) \quad = \quad 0.$$

We can use this property to kill terms in advance and this way keep the number of terms limited. Now we also go through the lattice in the rectangular way, rather than the diagonal way. This is shown in program 9:

```
#-
     #define SIZE "9"
     CFunctions  d,e,f,acc;
     Symbols x,x1,x2,x3,q,[q-1];
     PolyFun acc;
     *
     L  F = 1
     #do i = 0,'SIZE'^2-1
       #if ( {'i'%'SIZE'} != {'SIZE'-1} )
                 *e('i',{'i'+1})
       #endif
       #if ( {('i'/'SIZE')%'SIZE'} != {'SIZE'-1} )
                 *e('i',{'i'+'SIZE'})
       #endif
     #enddo
             ;
     .sort
     #do i = 0,'SIZE'^2-1
      id    e('i',x2?) = 1-d('i',x2);
      id  d('i',x?) = f(x);
      id  d(x?,'i') = f(x);
```

```
 if ( count(f,1) == 0 );
    Multiply acc([q-1]+1);
 else;
    repeat id f(x?)*f(x?) = f(x);
    ChainIn f;
    repeat id f(x1?,x2?,?a) = d(x1,x2)*f(x2,?a);
    id  d(x1?,x1?) = 1;
    repeat;
        id  d(x2?,x1?)*d(x3?,x1?) = d(x2,x3)*d(x3,x1);
        id  e(x1?,x2?)*d(x1?,x2?) = 0;
        id  d(x1?,x1?) = 1;
    endrepeat;
    repeat;
        id  d(x1?,x2?)*d(x1?,x3?) = d(x2,x3)*d(x1,x2);
        id  e(x1?,x2?)*d(x1?,x2?) = 0;
        id  d(x1?,x1?) = 1;
    endrepeat;
    id  f(x?) = 1;
 endif;
 B e;
 .sort:'i';
#enddo
PolyFun;
id  acc(x?) = x;
id [q-1] = q-1;
Print +f +s;
.end
```

The major improvement here is the amount of diskspace needed. We bracket in $e$ to make it such that the FORM compression causes the $e$ not to use much extra space. We could acchieve this also by changing the order of declaration of $e$ and $d$, but then the pattern matching takes more time, because it will start with the $e$'s and there are usually more of those. This means that there will be more work until FORM realizes that a match cannot occur. The results of this program are in table 9.

| N | Time (sec) | (Disk)space needed |
|---|---|---|
| 6 | 0.70 | 15 K |
| 7 | 3.30 | 57 K |
| 8 | 15.3 | 215 K |
| 9 | 67.7 | 807 K |
| 10 | 290 | 2.99 M |
| 11 | 1221 | 11.0 M |

Table 9: Execution times of program 9 on a Pentium 850

The next improvement, although only slightly, is by chaining the $d$ together. This can be done, because the only thing relevant is connectivity, not in which way connected $d$'s are actually connected. The actual connections were relevant in the previous program because we needed a particular $d$ with an $e$ to make them vanish. Once we have just the structures there are two advantages: all possibilities for making an $e$ vanish are included, and there are fewer $d$ functions. On the other hand the pattern matching has more wildcards and takes much more time. This all results in program 10:

```
#-
```

```
#define SIZE "8"
CFunctions  f,d,e,acc;
Symbols x,x1,x2,x3,q,[q-1];
PolyFun acc;
*
L  F = 1
#do i = 0,'SIZE'^2-1
  #if ( {'i'%'SIZE'} != {'SIZE'-1} )
            *e('i',{'i'+1})
  #endif
  #if ( {('i'/'SIZE')%'SIZE'} != {'SIZE'-1} )
            *e('i',{'i'+'SIZE'})
  #endif
#enddo
        ;
.sort
#do i = 0,'SIZE'^2-1
 id    e('i',x?) = 1-d('i',x);
 repeat id d('i',?a)*d('i',?b) = d('i',?a,?b);
 repeat id d(?a,x1?,?b)*d(?c,x1?,?d) = d(?a,?c,x1,?b,?d);
 Symmetrize d;
 repeat id d(?a,x1?,x1?,?b) = d(?a,x1,?b);
 id    d(?a,x1?,?b,x2?,?c)*e(x1?,x2?) = 0;
 id    d = 1;
 id,ifmatch->1,d('i',x1?) = 1;
 id,ifmatch->1,d('i',?b) = d(?b);
 Multiply acc([q-1]+1);
 Label 1;
 id    d = 1;
 B e;
 .sort:'i';
#enddo
PolyFun;
id  acc(x?) = x;
id [q-1] = q-1;
Print +f +s;
.end
```

The results of this program are in table 10. We notice a slight speedup over program 9.

| N | Time (sec) | (Disk)space needed |
|---|---|---|
| 6 | 0.69 | 15 K |
| 7 | 3.15 | 57 K |
| 8 | 14.3 | 215 K |
| 9 | 62.9 | 808 K |
| 10 | 267 | 2.99 M |
| 11 | 1121 | 11.0 M |
| 12 | 4599 | 39.9 M |

Table 10: Execution times of program 10 on a Pentium 850

To try the diagonal way of stepping through the lattice works less efficiently here. The fact that vertices that are adjacent to already treated vertices can be adjacent to each other helps very much in the elimination of terms.

It is possible to save a bit more time by changing from numbers in the functions $d$ and $e$ to indices and declare these functions as tensors. The reason is one of economy in FORM. It knows that tensors can have only indices (or vectors) as their arguments. In general function arguments can be nearly anything. Hence tensors use a little bit less space (which will not be relevant here because the internal compression makes it nearly invisible) and their pattern matching is much faster. The last thing can be noticed in the execution speeds. We also change the startup of the program to make it a truely multidimensional program (program 11).

```
#-
#define SIZE "11"
#define DIMENSION "2"
CFunctions  acc;
Tensors d,e,f;
Symbols x,q,[q-1];
Indices k,k0,...,k{`SIZE'^`DIMENSION'};
PolyFun acc;
Format nospaces;
Format 80;
*
L  F = 1
#do i = 0,`SIZE'^`DIMENSION'-1
#do j = 1,`DIMENSION'
  #if ( {(`i'/{`SIZE'^{`j'-1}})%`SIZE'} != {`SIZE'-1} )
            *e(k`i',k{`i'+`SIZE'^{`j'-1}})
  #endif
#enddo
#enddo
        ;
.sort
#do i = 0,`SIZE'^`DIMENSION'-1
  id    e(k`i',k?) = 1-d(k`i',k);
  repeat id d(k`i',?a)*d(k`i',?b) = d(k`i',?a,?b);
  repeat id d(?a,k?,?b)*d(?c,k?,?d) = d(?a,?c,k,?b,?d);
  Symmetrize d;
  repeat id d(?a,k?,k?,?b) = d(?a,k,?b);
  id    d(?a,k1?,?b,k2?,?c)*e(k1?,k2?) = 0;
  id,ifmatch->1,d(k`i',k?) = 1;
  id,ifmatch->1,d(k`i',?b) = d(?b);
    Multiply acc([q-1]+1);
  Label 1;
  id    d = 1;
  B e;
  .sort:`i';
#enddo
PolyFun;
id  acc(x?) = x;
id [q-1] = q-1;
Print +f +s;
.end
```

The program is now actually rather short, but yet fast. The execution times are in table 11.

A slight improvement can still be made. We had to declare $d$ before $e$, because in that case the statement

```
  id    d(?a,k1?,?b,k2?,?c)*e(k1?,k2?) = 0;
```

| N | Time (sec) | (Disk)space needed |
|---|---|---|
| 6 | 0.60 | 15 K |
| 7 | 2.68 | 57 K |
| 8 | 12.2 | 215 K |
| 9 | 53.7 | 807 K |
| 10 | 226 | 2.99 M |
| 11 | 947 | 11.0 M |
| 12 | 3941 | 39.8 M |
| 13 | 15396 | 400 M |

Table 11: Execution times of program 11 on a Pentium 850

would not be so costly. There are potentially many occurrences of $e$ and only very few of $d$. Hence if we can cut down on the possible matches in $e$ very quickly, we might declare $e$ before $d$ and hence not need the bracket statement for saving space. This way we can use the bracket statement as in program 2 when the lattice becomes so big that one term cannot keep the whole polynomial in q anylonger, or when we would like to work at finite temperature. We can make the required restriction with sets on the wildcarding:

```
id  d(?a,k1?,?b,k2?,?c)*e(k1?{k{`i'+1},...,k{`i'+`N'^{`D'-1}}}
        ,k2?{k{`i'+1},...,k{`i'+`N'^{`D'-1}}}) = 0;
```

in which the size of the lattice is $N$ and the dimension is $D$. This way the whole program (12) becomes

```
#-
*  D is the dimension and N is the size of the lattice
#define N "12"
#define D "2"
CFunctions acc;
Tensors e,d;
Symbols x,q,[q-1];
Indices k,k0,...,k{`N'^`D'+`N'^{`D'-1}-1};
PolyFun acc;
Format nospaces;
Format 80;
*
L   F = 1
#do i = 0,`N'^`D'-1
#do j = 1,`D'
  #if ( {(`i'/{`N'^{`j'-1}})%`N'} != {`N'-1} )
            *e(k`i',k{`i'+`N'^{`j'-1}})
  #endif
#enddo
#enddo
        ;
.sort
#do i = 0,`N'^`D'-1
  id  e(k`i',k?) = 1-d(k`i',k);
  repeat id  d(k`i',?a)*d(k`i',?b) = d(k`i',?a,?b);
  repeat id  d(?a,k?,?b)*d(?c,k?,?d) = d(?a,?c,k,?b,?d);
  Symmetrize d;
  repeat id  d(?a,k?,k?,?b) = d(?a,k,?b);
  id  d(?a,k1?,?b,k2?,?c)*e(k1?{k{`i'+1},...,k{`i'+`N'^{`D'-1}}}
```

```
            ,k2?{k{'i'+1},...,k{'i'+'N'^{'D'-1}}}) = 0;
    id,ifmatch->1,d(k'i',k?) = 1;
    id,ifmatch->1,d(k'i',?b) = d(?b);
      Multiply acc([q-1]+1);
    Label 1;
    id  d = 1;
    .sort:'i';
  #enddo
  PolyFun;
  id  acc(x?) = x;
  id  [q-1] = q-1;
  Print +f +s;
  .end
```

This program is actually slightly faster than program 11. The order of the declaration of $e$ and $d$ is now relevant for two reasons. It saves much diskspace and we put the set restrictions only inside the occurrence of $e$, not inside $d$. If we change the order of declaration the wildcarding will be more expensive because searching inside $d$ involves three argument wildcards and all possibilities will be tried. The execution times are now given in table 12.

| D | N | Time (sec) | (Disk)space needed |
|---|---|---|---|
| 2 | 6 | 0.59 | 15 K |
| 2 | 7 | 2.70 | 57 K |
| 2 | 8 | 11.9 | 215 K |
| 2 | 9 | 51.2 | 807 K |
| 2 | 10 | 214 | 2.99 M |
| 2 | 11 | 880 | 11.0 M |
| 2 | 12 | 3515 | 39.8 M |
| 3 | 3 | 13.8 | 212 K |

Table 12: Execution times of program 12 on a Pentium 850

Of course, once we realize that limiting the number of $e$ functions that can take place in the pattern matchings, we can do a little better by making sure the $e$'s belonging to vertices that cannot make a contribution are not present at all. This makes the wildcard set restriction on the first index superfluous. Of course the program becomes slightly more complicated this way (as should be for program 13):

```
#-
#define N "12"
#define D "2"
CFunctions acc;
Tensors e,d;
Symbols x,q,[q-1];
Indices k,k0,...,k{'N'^'D'+'N'^{'D'-1}-1};
PolyFun acc;
Off Statistics;
Format nospaces;
Format 80;
*
#do i = 0,'N'^'D'-1
L   F'i' = 1
#do j = 1,'D'
```

```
    #if ( {(`i'/{`N'^{`j'-1}})%`N'} != {`N'-1} )
              *e(k`i',k{`i'+`N'^{`j'-1}})
    #endif
  #enddo
        ;
  #enddo
  .sort
  On Statistics;
  Hide;
  L F = F0*...*F{`N'^{`D'-1}-1};
  #do i = 0,`N'^`D'-1
  #if ( {`i'+`N'^{`D'-1}} < {`N'^`D'} )
   Multiply F{`i'+`N'^{`D'-1}};
  #endif
    id  e(k`i',k?) = 1-d(k`i',k);
    repeat id  d(k`i',?a)*d(k`i',?b) = d(k`i',?a,?b);
    repeat id  d(?a,k?,?b)*d(?c,k?,?d) = d(?a,?c,k,?b,?d);
    Symmetrize d;
    repeat id  d(?a,k?,k?,?b) = d(?a,k,?b);
    id  d(?a,k1?,?b,k2?,?c)*
                e(k1?,k2?{k{`i'+1},...,k{`i'+`N'^{`D'-1}}}) = 0;
    id,ifmatch->1,d(k`i',k?) = 1;
    id,ifmatch->1,d(k`i',?b) = d(?b);
      Multiply acc([q-1]+1);
    Label 1;
    id  d = 1;
    .sort:`i';
  #enddo
  PolyFun;
  id  acc(x?) = x;
  id  [q-1] = q-1;
  Print +f +s;
  .end
```

The increase in speed is rather surprising as is shown in table 13.

| D | N | Time (sec) | (Disk)space needed | Computer |
|---|---|---|---|---|
| 3 | 3 | 12.63 | 212 K | P850 |
| 2 | 6 | 0.44 | 15 K | P850 |
| 2 | 7 | 1.80 | 57 K | P850 |
| 2 | 8 | 7.28 | 215 K | P850 |
| 2 | 9 | 29.6 | 807 K | P850 |
| 2 | 10 | 117.9 | 2.99 M | P850 |
| 2 | 11 | 473 | 11.0 M | P850 |
| 2 | 12 | 1860 | 39.8 M | P850 |
| 2 | 13 | 7324 | 400 M | P850 |
| 2 | 13 | 3676 | 400 M | P1700 |
| 2 | 14 | 14201 | 1599 M | P1700 |

Table 13: Execution times of program 13.

Actually, considering we are working on a lattice, we could do away with the matching with the $e$ functions completely if we can represent the future links in the form of sets that can be used as a restriction in the wildcarding. We will need one set for each dimension. This way program 14 becomes:

```
#-
#define N "7"
#define D "2"
CFunctions acc;
Tensors e,d;
Symbols x,q,[q-1];
Indices k,k0,...,k{`N'^`D'+`N'^{`D'-1}-1};
set kk0:k0,...,k{`N'^`D'-1};
#do d = 1,`D'
set kk`d':
#do i = 0,`N'^`D'-1
  #if ( {(`i'/{`N'^{`d'-1}})%`N'} != {`N'-1} )
 k{`i'+`N'^{`d'-1}}
  #else
 k
  #endif
#enddo
 ;
#enddo
Off Statistics;
Format nospaces;
Format 80;
*
#do i = 0,`N'^`D'-1
L   F`i' = 1
#do j = 1,`D'
  #if ( {(`i'/{`N'^{`j'-1}})%`N'} != {`N'-1} )
            *e(k`i',k{`i'+`N'^{`j'-1}})
  #endif
#enddo
        ;
#enddo
.sort
PolyFun acc;
On Statistics;
Hide;
L F = 1;
#do i = 0,`N'^`D'-1
  Multiply F`i';
  id  e(k`i',k?) = 1-d(k`i',k);
  repeat id  d(k`i',?a)*d(k`i',?b) = d(k`i',?a,?b);
  repeat id  d(?a,k?,?b)*d(?c,k?,?d) = d(?a,?c,k,?b,?d);
  Symmetrize d;
  repeat id  d(?a,k?,k?,?b) = d(?a,k,?b);
  id,ifmatch->1,d(k`i',k?) = 1;
  id,ifmatch->1,d(k`i',?b) = d(?b);
    Multiply acc([q-1]+1);
  Label 1;
  id  d = 1;
  #do d = 1,`D'
    id  d(?a,k1?kk0[x],?b,k2?kk`d'[x],?c) = 0;
  #enddo
```

```
    .sort:'i';
  #enddo
  PolyFun;
  id  acc(x?) = x;
  id  [q-1] = q-1;
  Print +f +s;
  .end
```

Note that now we need one statement per dimension because for each dimension we have a set. This gives even faster execution times as shown in table 14.

| D | N | Time (sec) | (Disk)space needed | Computer |
|---|---|---|---|---|
| 3 | 3 | 6.35 | 212 K | P850 |
| 2 | 6 | 0.22 | 15 K | P850 |
| 2 | 7 | 1.03 | 57 K | P850 |
| 2 | 8 | 4.42 | 215 K | P850 |
| 2 | 9 | 17.9 | 807 K | P850 |
| 2 | 10 | 73.4 | 2.99 M | P850 |
| 2 | 11 | 307 | 11.0 M | P850 |
| 2 | 12 | 1248 | 39.8 M | P850 |
| 2 | 13 | 5151 | 396 M | P850 |
| 2 | 14 | 9717 | 1604 M | P1700 |
| 2 | 15 | 37442 | 5971 M | P1700 |

Table 14: Execution times of program 14.

Yet another improvement can be made when one considers that all the action is for occurrences of $d$ that contain the vertex $i$. Hence by a small change of the order of the statements we get the central loop of program 15:

```
#do i = 0,'N'^'D'-1
  Multiply F'i';
  repeat id  d(?a,k?,?b)*d(?c,k?,?d) = d(?a,?c,k,?b,?d);
  Symmetrize d;
  repeat id  d(?a,k?,k?,?b) = d(?a,k,?b);
  #do d = 1,'D'
    id  d(k'i',?a,k1?kk0[x],?b,k2?kk'd'[x],?c) = 0;
  #enddo
  id,ifmatch->1,d(k'i',k?) = 1;
  id,ifmatch->1,d(k'i',?b) = d(?b);
    Multiply acc([q-1]+1);
  Label 1;
  id  d = 1;
  .sort:'i';
#enddo
```

For the bigger lattices this results in a 10 percent savings as can be seen in table 15.

Because it is rather hard to improve the program at this point (more mathematical input about the polynomials might be called for), we can extend it a bit by allowing lattices that are not $N^D$. First the two dimensional version which is program 16:

```
  #-
  #define N1 "7"
  #define N2 "8"
```

| D | N | Time (sec) | (Disk)space needed | Computer |
|---|---|---|---|---|
| 3 | 3 | 5.56 | 211 K | P850 |
| 2 | 6 | 0.24 | 15 K | P850 |
| 2 | 7 | 0.95 | 57 K | P850 |
| 2 | 8 | 3.89 | 215 K | P850 |
| 2 | 9 | 16.2 | 807 K | P850 |
| 2 | 10 | 66.2 | 2.99 M | P850 |
| 2 | 11 | 275 | 10.8 M | P850 |
| 2 | 12 | 1113 | 39.8 M | P850 |
| 2 | 13 | 4589 | 396 M | P850 |
| 2 | 14 | 18557 | 1605 M | P850 |
| 2 | 12 | 560 | 39.8 M | P1700 |
| 2 | 13 | 2264 | 396 M | P1700 |
| 2 | 14 | 8920 | 1605 M | P1700 |
| 2 | 15 | 34407 | 5.97 G | P1700 |

Table 15: Execution times of program 15. The fact that the factor between the P850 and the P1700 is not exactly a constrant reflects the different architecture of the machines.

```
#define D "2"
CFunctions acc;
Tensors e,d;
Symbols x,q,[q-1];
Indices k,k0,...,k{`N1'*`N2'+`N1'-1};
set kk0:k0,...,k{`N1'*`N2'-1};
set kk1:
#do i = 0,`N1'*`N2'-1
  #if ( {`i'%`N1'} != {`N1'-1} )
 k{`i'+1}
  #else
 k
  #endif
#enddo
 ;
set kk2:
#do i = 0,`N1'*`N2'-1
  #if ( {(`i'/`N1')%`N2'} != {`N2'-1} )
 k{`i'+`N1'}
  #else
 k
  #endif
#enddo
 ;
Off Statistics;
Format nospaces;
Format 80;
*
#do i = 0,`N1'*`N2'-1
L   F`i' = 1
  #if ( {`i'%`N1'} != {`N1'-1} )
            *e(k`i',k{`i'+1})
  #endif
```

```
  #if ( {(`i'/`N1')%`N2'} != {`N2'-1} )
             *e(k`i',k{`i'+`N1'})
  #endif
        ;
#enddo
id  e(k?,k1?) = 1-d(k,k1);
repeat id  d(k?,?a)*d(k?,?b) = d(k,?a,?b);
.sort
PolyFun acc;
On Statistics;
Hide;
L F = 1;
#do i = 0,`N1'*`N2'-1
  Multiply F`i';
  repeat id  d(?a,k?,?b)*d(?c,k?,?d) = d(?a,?c,k,?b,?d);
  Symmetrize d;
  #do d = 1,`D'
    id  d(k`i',?a,k1?kk0[x],?b,k2?kk`d'[x],?c) = 0;
  #enddo
  repeat id  d(?a,k?,k?,?b) = d(?a,k,?b);
  id,ifmatch->1,d(k`i',k?) = 1;
  id,ifmatch->1,d(k`i',?b) = d(?b);
    Multiply acc([q-1]+1);
  Label 1;
  id  d = 1;
  .sort:`i';
#enddo
PolyFun;
id  acc(x?) = x;
id  [q-1] = q-1;
Print +f +s;
.end
```

This gives some interesting results when comparing $i \times j$ with $j \times i$ as shown in table 16. Clearly,

| $N_1$ | $N_2$ | Time (sec) | (Disk)space needed |
|---|---|---|---|
| 4 | 8 | 0.05 | 3074 |
| 8 | 4 | 0.75 | 50570 |

Table 16: Execution times of program 16 on a P850.

one should put the smaller size first, because this leaves the smallest number of untreated vertices in the intermediate expressions.

The three dimensional version of program 16 is program 17:

```
#-
#define N1 "4"
#define N2 "3"
#define N3 "3"
#define D "3"
CFunctions acc;
Tensors e,d;
Symbols x,q,[q-1];
Indices k,k0,...,k{`N1'*`N2'*`N3'+`N1'*`N2'-1};
set kk0:k0,...,k{`N1'*`N2'*`N3'-1};
```

```
set kk1:
#do i = 0,`N1'*`N2'*`N3'-1
  #if ( {`i'%`N1'} != {`N1'-1} )
 k{`i'+1}
  #else
 k
  #endif
#enddo
 ;
set kk2:
#do i = 0,`N1'*`N2'*`N3'-1
  #if ( {(`i'/`N1')%`N2'} != {`N2'-1} )
 k{`i'+`N1'}
  #else
 k
  #endif
#enddo
 ;
set kk3:
#do i = 0,`N1'*`N2'*`N3'-1
  #if ( {(`i'/{`N1'*`N2'})%`N3'} != {`N3'-1} )
 k{`i'+`N1'*`N2'}
  #else
 k
  #endif
#enddo
 ;
Off Statistics;
Format nospaces;
Format 80;
*
#do i = 0,`N1'*`N2'*`N3'-1
L   F`i' = 1
  #if ( {`i'%`N1'} != {`N1'-1} )
            *e(k`i',k{`i'+1})
  #endif
  #if ( {(`i'/`N1')%`N2'} != {`N2'-1} )
            *e(k`i',k{`i'+`N1'})
  #endif
  #if ( {(`i'/{`N1'*`N2'})%`N3'} != {`N3'-1} )
            *e(k`i',k{`i'+`N1'*`N2'})
  #endif
        ;
#enddo
id  e(k?,k1?) = 1-d(k,k1);
repeat id  d(k?,?a)*d(k?,?b) = d(k,?a,?b);
.sort
PolyFun acc;
On Statistics;
Hide;
L F = 1;
#do i = 0,`N1'*`N2'*`N3'-1
  Multiply F`i';
  repeat id  d(?a,k?,?b)*d(?c,k?,?d) = d(?a,?c,k,?b,?d);
```

```
  Symmetrize d;
  #do d = 1,‘D’
    id  d(k‘i’,?a,k1?kk0[x],?b,k2?kk‘d’[x],?c) = 0;
  #enddo
  repeat id  d(?a,k?,k?,?b) = d(?a,k,?b);
  id,ifmatch->1,d(k‘i’,k?) = 1;
  id,ifmatch->1,d(k‘i’,?b) = d(?b);
    Multiply acc([q-1]+1);
  Label 1;
  id  d = 1;
  .sort:‘i’;
#enddo
PolyFun;
id  acc(x?) = x;
id  [q-1] = q-1;
Print +f +s;
.end
```

Here the differences in execution time and needed space become even more striking. This is shown in table 17. The execution of the configuration $4 \times 4 \times 3$ we have not attempted. It would

| $N_1$ | $N_2$ | $N_3$ | Time (sec) | (Disk)space needed |
|---|---|---|---|---|
| 3 | 3 | 3 | 5.56 | 211 K |
| 3 | 3 | 4 | 18.7 | 718 K |
| 3 | 3 | 5 | 40.7 | 1.50 M |
| 3 | 3 | 6 | 68.8 | 2.49 M |
| 3 | 4 | 3 | 404 | 12.1 M |
| 4 | 3 | 3 | 469 | 12.3 M |
| 3 | 4 | 4 | 2670 | 168 M |
| 3 | 4 | 5 | 7659 | 516 M |
| 3 | 4 | 7 | 24132 | 1.94 G |

Table 17: Execution times of program 17 on a P850.

take several days on the P850 computer and involve more than 12 Gbytes of diskspace as can be derived from a run of the $4 \times 4 \times 4$ lattice on a bigger computer.

It is not excluded that more improvements can be made. The main parameter to look for is the number of possible terms in the intermediate results. One could also try to find a better representation for the polynomials. In the later stages of the program most time goes to the polynomial manipulations. Another place to look for improvement is in the pattern matching of the $d$ tensors. Because of the large number of possibilities FORM spends much time when deciding that there is no match.

Conclusions: With the last program, either in its two or in its three dimensional version, there should be no problem extending the currently available results significantly.