Introduction to FORM

Jos Vermaseren

Part 2: Substitutions

The most powerful command in FORM is the substitution. It allows the replacement of one object by another in ways that depend on the contents of the object.
The command is called the 'id' or 'identify' statement. This name originates in Schoonschip. Let us have a look at a very simple example:

```
    Symbols a,b,c,d;
    Off statistics;
    Local F = (a+b)^2;
    Print;
    .sort
   F =
      b^2 + 2*a*b + a^2;
    id b = c+d;
    Print;
    .end
   F =
      d^2 + 2*c*d + c^2 + 2*a*d + 2*a*c + a^2;
```

We see that the 'id' statement has a left hand side (LHS) and a right hand side (RHS) separated by an equals sign. The LHS is to be replaced by the RHS.

The order of operation in the ‘id’ statement is that the LHS, also called the pattern, is matched as many times as possible. Only after that the RHS is substituted. After that FORM continues to the next statement:

```
Symbol x;
Off Statistics;
Local expr = x + 1/x;
id x = x+1;
Print;
.sort

expr =
   1 + x^-1 + x;
```

We see here that the substitution was indeed made once, but only on positive powers of x. This finds its origins in the fact that composite denominators are a bit of a weakness in FORM. They are arguments of a special internal function and hence normally treated as function arguments. When we make a regular substitution, FORM doesn’t look inside function arguments. We will see later how to do that.

Let us continue with the example. If we put another id-statement of the same type we can now predict what the result will be:

```
id x = x+1;
Print;
.end
```

```
expr =
   2 + x^-1 + x;
```

This result can of course also be obtained in a single module:

```
Symbol x;
Off Statistics;
Local expr = x + 1/x;
id x = x+1;
id x = x+1;
Print;
.end

 expr =
    2 + x^-1 + x;
```

Let us make the example a bit more complicated now. We have seen the `sum_` function before.

```
 Symbols x,y,z,k;
 Local expr = sum_(k,-2,5,x^k);
 Print;
 .sort
expr =
   1 + x^-2 + x^-1 + x + x^2 + x^3 + x^4 + x^5;
 id,x^2 = y;
 Print;
 .sort
expr =
   1 + x^-2 + x^-1 + x*y + x*y^2 + x + y + y^2;
 id  x*y = z;
 Print;
 .sort
expr =
   1 + x^-2 + x^-1 + x + y*z + y + y^2 + z;
 id  1/x = z;
 Print;
 .end
expr =
   1 + x + y*z + y + y^2 + 2*z + z^2;
```

We see that we can replace powers of an object. It will then only take integer multiples of this power.

We can also substitute combinations of symbols.

And to substitute negative powers we specify `1/x`

The next example is a bit more complicated. If we use an expression in the RHS of a substitution, it will substitute the expression as it is at the start of the mudule.

```
Symbol x,y;
Local expr = x*y;
id  x = expr;
Print;
.sort

expr =
   x*y^2;

id  x = expr;
id  x = expr;
Print;
.end

expr =
   x*y^6;
```

One can see here that we get `x*y^6`, because at the start of the last module expr is `x*y^2`.

Now we use the id-statement to generate the 19-th Fibonacci number. This involves basically the statement `id x^2 = x+1;`. However we don't want to apply it just once, but rather as many times as needed to reduce the expression to terms that have either zero or one power of x. This is done with the 'repeat' environment. FORM will execute its contents, and if, after reaching the endrepeat statement, any changes occurred, the entire contents will be executed again.

```
    Symbol x;
    Local Fibonacci19 = x^18;
    repeat;
      id x^2 = x+1;
    endrepeat;
    id  x = 1;
    Print;
    .end
Time =        0.00 sec    Generated terms =         61
      Fibonacci19         Terms in output =          1
                          Bytes used      =         10

   Fibonacci19 =
      4181;
```

The repeat facility is very important!

Here we see an example of the use of id- and repeat-statements with non-commuting objects. Those have to be declared as functions.

```
 Symbols hbar,m;
 Functions x,p,H;
 Local [H,x] = H*x - x*H;
 id  H = p^2/(2*m);
 Print;
 .sort
[H,x] =
   - 1/2*x*p*p*m^-1 + 1/2*p*p*x*m^-1;
 repeat;
   id x*p = p*x + hbar*i_;
 endrepeat;
 Print;
 .end
[H,x] =
   - p*i_*hbar*m^-1;
```

Notice that `i_` is the imaginary i. Notice also the name of the expression being `[H,x]`, just to aid the eye.

In the next program we play a bit with sigma matrices. First we define the products of two different matrices:

```
Function s;
Index k;
Dimension 3;
Local [s(1)*s(2)] = i_*e_(1,2,3)*e_(1,2,k)*s(k);
Local [s(1)*s(3)] = i_*e_(1,2,3)*e_(1,3,k)*s(k);
Local [s(2)*s(3)] = i_*e_(1,2,3)*e_(2,3,k)*s(k);
Contract;
Print;
.sort
```

```
[s(1)*s(2)] =
   s(3)*i_;

[s(1)*s(3)] =
    - s(2)*i_;

[s(2)*s(3)] =
   s(1)*i_;
```

```
Local F = ( s(1)*s(2) + s(1) + s(2) + s(3) )^4;
repeat;
  id s(2)*s(1) = -s(1)*s(2);
  id s(3)*s(1) = -s(1)*s(3);
  id s(3)*s(2) = -s(2)*s(3);
  id s(1)*s(2) = [s(1)*s(2)];
  id s(1)*s(3) = [s(1)*s(3)];
  id s(2)*s(3) = [s(2)*s(3)];
  id s(1)^2 = 1;
  id s(2)^2 = 1;
  id s(3)^2 = 1;
endrepeat;
Print F;
.end
 F =
    8*i_;
```

Next we define the expression F as some messy combination of those matrices. The repeat loop contains all the statements to reduce the terms to something simpler. We use the anti-commutation rules, the previously defined products and the rules for the squares.

Eventually there will be an unique simplest answer.

Now we will have a look at generic patterns. If we want to say: "for any x", we write **x?**. We call **x?** a wildcard.

```
Symbols x,y,z,n;
Local F = x^2 + y^3 + 1;
id x? = z;
Print;
.sort
F =
  1 + z^2 + z^3;
```

The id-statement here says: replace any symbol by the symbol z, and we can see that this is indeed what FORM does.

When we continue, we say: "substitute any power of z by x". In this case there is also a match for zero powers.

Note that the true mathematician may say: "but this is poorly defined, because on for instance `z^10` it could give any number of matches." The rule in FORM is that there will be a single fitting of the power n that should be taken.

```
    id  z^n? = x;
    Print;
    .sort
   F =
      3*x;
    Local G = F + y^2 + 1;
    id  x?^n? = z;
    Print G;
    .end
   G =
      1 + 4*z;
```

Finally, `x?^n?` needs a symbol and hence will not match on zero powers. This is a FORM convention.

In this example we see the use of wildcards as function arguments. It is also a different way to calculate Fibonacci numbers.

```
    Symbols last, secondlast, dummy;
    Function F;
    Local Fibonacci19 = F(1,1) * dummy^17;
    repeat;
      id F(last?,secondlast?)*dummy = F(last+secondlast,last);
    endrepeat;
    id  F(last?,secondlast?) = last;
    Print;
    .end
Time =       0.00 sec    Generated terms =          1
      Fibonacci19        Terms in output =          1
                         Bytes used      =         10

   Fibonacci19 =
      4181;
```

When a wildcard is used in the RHS, it doesn't need the question mark. It does however get the identity that the LHS wildcard obtains during the pattern matching. We will see that better in some of the following examples.

The next way to calculate Fibonacci numbers looks more like the original recursive definition, but we see that it generates many terms and hence starts costing a measurable amount of time.

```
Symbols n;
Function F;
On statistics;
Local Fibonacci19 = F(19);
repeat;
  id F(1) = 1;
  id F(2) = 1;
  id F(n?) = F(n-1)+F(n-2);
endrepeat;
Print;
.end
```

```
Time =        0.02 sec    Generated terms =        4181
      Fibonacci19         Terms in output =           1
                          Bytes used      =          10

   Fibonacci19 =
      4181;
```

One of the things we should learn from this is that the choice of algorithm can make a big difference. Experimentation can pay off handsomely!

Usually there are many different ways in which things can be done in FORM. Which one is the most efficient depends on the problem and may depend on the selected notation.

Next we make the pattern slightly more complicated again. Here we differentiate a polynomial in two variables with respect to both variables in one statement.

```
Symbols x,y,z,m,n;
Local P = x^2*y^3 + x^3 + x^4*y^4;
Print;
.sort

P =
   x^2*y^3 + x^3 + x^4*y^4;

id x^m?*y^n? = m*x^(m-1)*n*y^(n-1);
Print;
.end

P =
   6*x*y^2 + 16*x^3*y^3;
```

When two identical wildcard objects occur in the pattern, they should be matched by the same object, as can be seen in the example below:

```
Vector x;
Tensors a;
Indices i,j,k;
Local F = a(i,j) * a(j,k) * a(k,x);
id  a(i?,k?)*a(k?,j?) = a(i,j);
Print;
.end

  F =
     a(i,k)*a(k,x);
```

Of course, when there are two possible, but mutually exclusive, possibilities for the match, it is not specified in the language of FORM which match will be taken. Hence in future versions there can be a change in which one is taken, if this is judged necessary for improving the pattern matching. One should write programs that are independent of such internal choices.

Usually of course one wants a complete reduction as below:

```
Vector x;
Tensors a;
Indices i,j,k;
Local F = a(i,j) * a(j,k) * a(k,x);
repeat;
  id  a(i?,k?)*a(k?,j?) = a(i,j);
endrepeat;
Print;
.end

 F =
    a(i,x);
```

In this case the order in which the process took place is irrelevant.

Making substitutions inside function arguments can be done in different ways. Which way is better depends on the problem. Below we show the generic way:

```
CFunction f;
Symbols x,y;
Local expr = f(x,f(x));
id x = y;
Print;
.sort
```

```
expr =
   f(x,f(x));
```

We see here that the substitution of x by y is not carried out inside the function arguments.

To go inside the function arguments we need to enter the argument environment. It can have parameters to specify which function(s) and/or which argument(s), but this can be looked up in the manual. We continue with our example:

```
Argument;
  id x = y;
EndArgument;
Print;
.sort

 expr =
    f(y,f(x));
```

This time the x in the first argument has indeed be replaced. However the x in the second argument hasn't been replaced as it was an argument inside an argument.

In this case we need to nest the argument environment as show below:

```
Argument;
  Argument;
    id x? = y^2;
  EndArgument;
EndArgument;
Print;
.end

expr =
   f(y,f(y^2));
```

There is a way to make a replacement of all occurrences of a given variable in the complete term. This is done with the `replace_` function.

```
CFunction f;
Symbols x,y;
Local expr = f(x,f(x));
Multiply replace_(x,y);
Print;
.end
```

```
expr =
   f(y,f(y));
```

This function should have an even number of arguments. The first argument of each pair is the object to be replaced, while the second argument is the object by which it should be replaced. The object to be replaced should be a single symbol, vector, index, function or tensor. There are some extra reasonable restrictions like 'an index cannot be replaced by a symbol' etc.

Using the `replace_` function one can make exchanges of variables in a rather simple way:

```
CFunction f;
Symbols x,y;
Local expr = x^3*y+f(x,f(x),y);
Multiply replace_(x,y,y,x);
Print;
.end

expr =
   x*y^3 + f(y,f(y),x);
```

All substitutions in a single replace function are executed at the same time. If there is more than one replace function, it is not specified which one will be executed first. Try to avoid this situation.

One can also use a wildcard function as in

```
Symbols x,y,z;
CFunctions f,g,h;
Local expr1 = f(x) + g(y);
id  f?(x) = h(x);
Print;
.sort

expr1 =
   g(y) + h(x);

Local expr2 = f(x) + g(y);
id  f?(x?) = z;
Print expr2;
.sort

expr2 =
   2*z;
```

```
Local expr3 = f(x+y) + f(x,y);
id  f?(x?) = z;
Print expr3;
.end
```

```
expr3 =
   z + f(x,y);
```

In the last case we see that the pattern matched the first term, as there the function had just a single argument. In the second term the function has two arguments, and hence there is no match.

At times it is necessary to have a wildcard that can contain an unspecified number of function arguments. Such wildcards are called argument field wildcards and are indicated by a questionmark, followed by an alphabetic character and possibly followed by one or more alphanumeric characters. Argument field wildcards should also have the questionmark in the RHS of the substitution:

```
Symbols x,y;
CFunctions f,g;
Local F = f(x,y,x) + f;
id  f(?a) = g(0,?a,0,?a,0);
Print;
.end
```

```
F =
   g(0,x,y,x,0,x,y,x,0) + g(0,0,0);
```

In the second term ?a matches an empty argument field. Hence in the substitution it gives no extra arguments.

Things become a bit more complicated when functions have special properties. One can declare functions or tensors to be symmetric, antisymmetric, cyclic and rcyclic (also symmetric under reversion the order of the arguments). These names are of course case insensitive.

```
    Symbols w,x,y,z;
    Indices W,X,Y,Z;
    CFunction F;
    Tensor S(symmetric), C(Cyclic);
    Local expr = F(x,y,z) + S(X,Y,Z) + C(X,Y,Z);
    id  F(?a,w?,?b) = F(w,0,?a,0,?b);
    id  S(?a,W?,?b) = S(W,0,?a,0,?b);
    id  C(?a,W?,?b) = C(W,0,?a,0,?b);
    Print;
    .end
   expr =
      F(z,0,x,y,0) + S(X,Y,Z) + C(0,0,Y,Z,X);
```

We see that in the first function, FORM tries first the match that involves ?b (the second wildcard) to be minimal and ?a (the first wildcard) to be maximal. Try to figure out what exactly happened with the cyclic function.

In a symmetric function one cannot use more than a single argument field wildcard. The combinatoric possibilities when several functions with the same argument field wildcards involved would become too complicated. Already with a single argument field wildcard there are n! possible matches when n arguments are involved.

For the next example we introduce a new shorthand notation. If the contents of a repeat/endrepeat environment is just a single statement one can use

```
repeat statement;
```

instead of

```
repeat;
  statement;
endrepeat;
```

Next is an example of how to define ones own gamma matrices and string them together into a trace.

```
CFunction g, Tr;
Vectors p1,p2,p3,p4,p5,p6;
Indices i1,i2,i3,i4,i5,i6;
Local expr = g(i1,i3,p1)*g(i3,i5,p2)*g(i5,i2,p3)
            *g(i2,i6,p4)*g(i6,i4,p5)*g(i4,i1,p6);
repeat id g(i1?,i2?,?a)*g(i2?,i3?,?b) = g(i1,i3,?a,?b);
Print;
.sort

expr =
   g(i1,i1,p1,p2,p3,p4,p5,p6);

id g(i1?,i1?,?a) = Tr(?a);
Print;
.end

expr =
   Tr(p1,p2,p3,p4,p5,p6);
```

Of course FORM has its own built in gamma matrices named `g_` and rather efficient trace algorithms, but we will study those at a later stage in a chapter that is particle physics specific.