SFB/CPP-06-15
TTP06-12
NIKHEF 06-002

Extension of the functionality of the symbolic program FORM by external software

M. Tentyukov [a,1] and J.A.M. Vermaseren [b]

[a] *Institut für Theoretische Teilchenphysik, Universität Karlsruhe D-76131 Karlsruhe, Germany*

[b] *NIKHEF, Kruislaan 409, 1098 SJ, Amsterdam, The Netherlands*

Abstract

We describe the implementation of facilities for the communication with external resources in the Symbolic Manipulation System FORM. This is done according to the POSIX standards defined for the UNIX operating system. We present a number of examples that illustrate the increased power due to these new capabilities.

1 Introduction

FORM [1,2] is a Symbolic Manipulation System (SMS) specialized to handle extremely large [2] expressions of many millions of terms in an efficient way. FORM allows mainly local operations on single terms, like replacing parts of a term or multiplying something to it (the "locality principle", see Sect.2.2). This special property of FORM allows it to deal with expressions that are much larger than the main memory available. It also enables parallelization [3]. Together these features make FORM a unique tool suitable for state of the art evaluations. It is widely used in Quantum Field Theory, where the calculation of very large sequences of algebraic expressions is required, e.g. [4–14].

[1] On leave from Joint Institute for Nuclear Research, 141980 Dubna, Moscow Region, Russian Federation.

[2] The notion of large is of course time dependent. Currently we mean with this expressions that can have a number of terms of the order $\mathcal{O}(10^8)$ or more. This exceeds the current capabilities of Mathematica of Maple by a large factor.

Effective manipulation of large expressions requires that all algebraic instructions are applied to a big sequence of terms. The sheer size of the intermediate results prevents storage of more than a single version of an expression. Hence in FORM there is no benefit in "interactivity" and it is used in an non-interactive way. FORM provides a special programming language adapted for the manipulating of large sequences of algebraic terms and the user supplies programs written in this language. The major non-local operation which brings expressions to a standard form (the sort operation) makes the language powerful enough to program quite non-trivial algorithms; several FORM packages are widely used in multiloop calculations, e.g. MINCER [15] and MATAD [16].

Because FORM programs mainly describe the treatment of individual terms, rather than that of complete expressions, and because there are few libraries for many popular algebraic operations, FORM is considered to be less user-friendly than for instance the Computer Algebra Systems (CAS) Mathematica or Maple. On the other hand, very large expressions and time consuming operations often need special rather than generic algorithms anyway and one cannot use the libraries in that case. There are however cases in which it is useful to have access to the facilities of other specialized software. That is why sometimes FORM is used in combination with other software systems which provide a more fluent control flow in conjunction with some specialized programs. An example is the combination of the programs QGRAF [17], Q2E and EXP [18] and MATAD glued together with the help of the Make utility.

Several other systems use FORM as a sub-system for the evaluation of "hard" expressions. The C-program DIANA [19,20] designed as a master program for higher-order calculations uses FORM (see implementations e.g. [21,22] and references in [20]). The automatic calculation systems CompHEP [23,24] and GRACE [25] are currently migrating from REDUCE to FORM for analytical calculations. SANC [26] is a recently developed world-wide network system which consists of SANC Servers and SANC Clients which uses FORM for analytical evaluations. Also FormCALC [27] uses Mathematica as a user-friendly front-end to FORM.

As one can see, FORM is often used in co-operation with other software. This makes the improvement of the FORM abilities to communicate with other programs into an urgent problem.

The locality principle restricts the variety of possible algorithms or at least the way they should be programmed. It is impossible to avoid non-local operations completely. Hence FORM supports a number of non-local operations the most important of which is the sort operation. Another is the explicit bracketing of some common factor. By implementing a general interface for FORM to communicate with external programs, other non-local operations like polynomial factoring or the elimination of Greatest Common Divisors (GCD) between numerators and denominators can be delegated to other (less efficient but less restricted) programs. Recently, such an interface was elaborated. The interface permits FORM to run external programs and to be embedded in other applications. In the present paper we concentrate on the detailed description of the interface.

Of course, the systematic introduction to the FORM language is far beyond the scope of this paper. There are different manuals and tutorials devoted the FORM language, see the FORM homepage [2].

After a very short introduction to the FORM language and the mechanisms behind it in Sect. 2.1, we discuss in Sect. 3 different ways to communicate with the underlying operating system (UNIX) available in FORM3.1 (the current FORM version). The new mechanism available in the upcoming version FORM3.2 permitting a dialog with an external program is presented in Sect. 3.3. The exact syntax is described in Sect. 4.2. A working example of using the REDUCE [31] system from FORM is described and discussed in Appendix E.

2 General concepts

2.1 Common structure

FORM is used non-interactively by executing a program that contains several parts called *modules*. The programs are executed module by module. Modules are terminated with "dot"-instructions that cause the execution of the module, see the example on the left of Fig.1.

This example consists of only two modules. There are two "dot"-instructions: a `.sort` and a `.end`. In both cases the result is sorted. `.end` additionally terminates the program.

The execution of each module is divided into three steps:

- **Compilation:** the input is translated into an internal representation.
- **Generating:** for each term of the input expressions the statements of the module are executed. This in general generates a lot of terms for each input term.
- **Sorting:** all the output terms that have been generated are sorted and equivalent terms are summed.

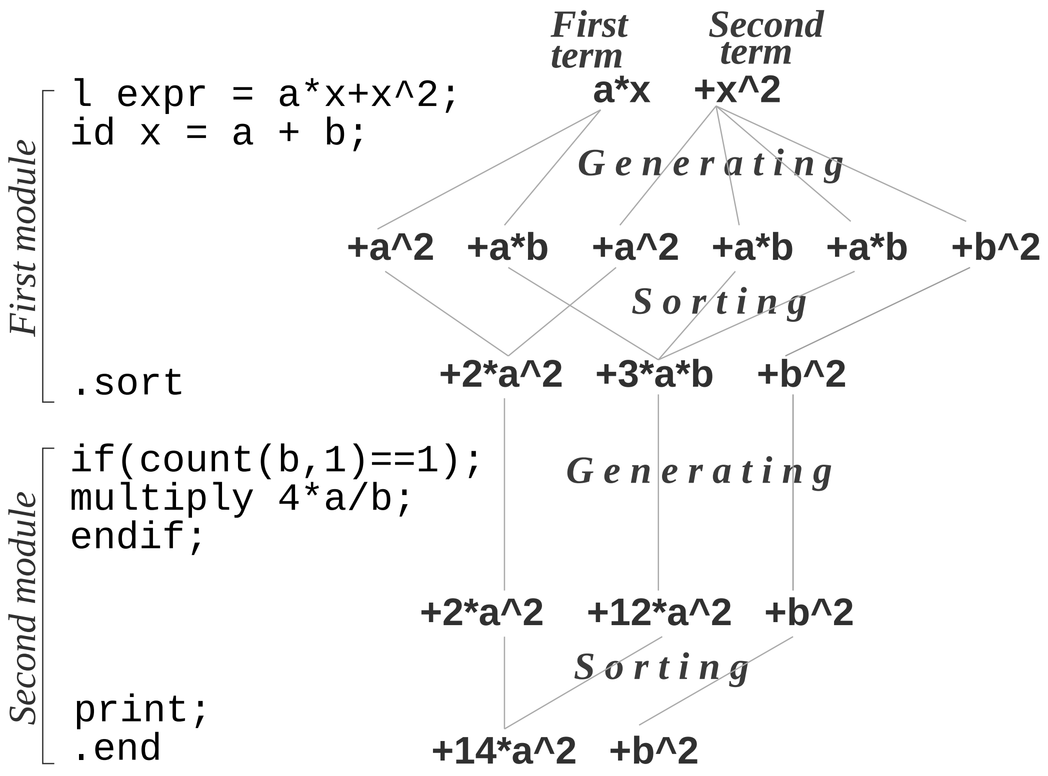

Fig. 1. A fragment of a typical FORM program. In the first module the expression $expr = ax + x^2$ is introduced, and then the substitution $x \to a + b$ is performed. In the second module only terms in which the degree of b is exactly 1 are multiplied by $4a/b$ (there is only one such a term in the expression).

2.2 The locality principle

Mostly FORM allows local operations on single terms, like replacing parts of a term or multiplying something to it; non-local operations like replacing a sum of two terms by another term are not allowed. We refer to this property as the *locality principle*: all *explicit* algebraic operations are local. Non-local operations are allowed only implicitly in the sorting procedure at the end of the modules, when equivalent terms are summed up, and in some special references to the contents of brackets in the input expression. Together with a sophisticated pattern matcher, this at first strong limitation allows the formulation of general and efficient algorithms. The limitation to local operations makes it possible to handle expressions as “streams” of terms that can be read sequentially from memory or a file and processed independently. This allows dealing with expressions that are larger than the available main memory and in addition it allows parallelism [3].

In principle non-local operations like polynomial factoring or GCD contraction could be performed at the level of the preprocessor, but this means that we would like to design some new CAS in the frame of the FORM preprocessor. The more natural solution would be to delegate this problem to other (less efficient but less restricted) CAS. At this point we come to the problem of how to interact with the outside world.

2.3 The control flow

The generation of terms is performed by the algebraic processor applying the algebraic imperatives (the statements inside a module) to each term of all active expressions. Thus, the control flow immediately splits into many independent branches. The control flow can be manipulated by restricting the application of statements to those terms that fulfil some criteria as is done in the following `if` statement

```
if ( count(b,1) == 1 );
  multilpy 4*a/b;
endif;
```

The above means that we assign to each power of b a degree of 1 and only terms in which the total degree is exactly 1 are multiplied by $4a/b$. Control flow based on complete expressions is not possible at the algebraic processor level. But it is possible at the preprocessor level.

The preprocessor in FORM is a notably important ingredient. It coordinates the execution of the program and eventually calls the processor to perform the actual module execution. All the "real" control flow in FORM is performed by the preprocessor. And, of course, interactivity is possible only at the preprocessor level. The interactive mode "Human – FORM" makes no sense because FORM usually deals with very big expressions, and intermediate results that may take Gigabytes of storage are not readily open for inspection by the programmer. On the other hand the interaction "FORM – Another program" often does make sense. In principle, a realization of this could be delegated to the underlying operating system. Of course, this introduces operating system dependent code and should be used with great care.

3 Interaction with the outside world

3.1 Running external programs in FORM3

Let us first look at the existing mechanisms of running external programs in FORM3. The easiest model of interaction is just to start some operating system command and wait until it finishes. The preprocessor instruction

```
#system systemcommand
```

forces a system command to be executed by the operating system. The complete string (excluding initial blanks or tabs) is passed to the operating system. FORM will then wait until control is returned.

One evident drawback of this mechanism is that there is no feedback from the system command. The only way to communicate is by means of the file system. It is quite clear that any nontrivial model would require that the output of the program is intercepted by FORM.

There is another preprocessor instruction, namely, `#pipe`:

```
#pipe systemcommand
```

This forces a system command to be executed by the operating system. The complete string (excluding initial blanks or tabs) is passed to the operating system. Next FORM will intercept the output of the command and read it as an input similar to the `#include` instruction. Whenever output is produced FORM will take action, and it will wait when no output is ready. After the command has been finished, FORM will continue with the next line.

This preprocessor instruction is much more powerful than the former one. But the communication channel established by the `#pipe` instruction doesn't satisfy all our requirements. Evidently, this is mostly a *unidirectional* channel; FORM could read an output, but establishing a dialog with the running program is very tricky (see Appendix C).

Let us now try to formulate the requirements more precisely.

3.2 Desired properties of the communication channel

Even the simple `#system` can be used for the extension of the FORM functionality. For example, consider an expression "`withGCD`" which is a ratio of two polynomials in d:

$$\frac{2d^4 + 3d^3 - 22d^2 - 13d + 30}{d^3 - 11d + 10}. \tag{1}$$

Factoring both the numerator and the denominator we get

$$\frac{(d^2+d-10)(2d+3)(d-1)}{(d^2+d-10)(d-1)}$$

so after contracting the GCD we obtain

$$2d+3. \tag{2}$$

Suppose, there is some light-weight CAS which is able to do the GCD contraction. How could we proceed with the `#system` instruction?

Consider the following simplified FORM program (for the full working example see Appendix B.1):

```
#define cmd . . .
Symbol d;
Local withGCD = (2*d^4+3*d^3-22*d^2-13*d+30)/(d^3-11*d+10);
#write <finput> "%E",withGCD
#system cat finput | `cmd' > foutput
Local noGCD =
#include foutput
        ;
print;
.end
```

Here we assume that the preprocessor variable `` `cmd' `` contains the command starting the external program.

First, we save the initial expression into the file "finput", then we run the external program performing the GCD contraction saving the result into another file "foutput" and at the end we read the result from the file. This works, and the result is given in the following simplified listing:

```
. . .
  withGCD =
    30/(10-11*d+d^3)-13/(10-11*d+d^3)*d
   -22/(10-11*d+d^3)*d^2+3/(10-11*d+
    d^3)*d^3+2/(10-11*d+d^3)*d^4;

  noGCD =
    3 + 2*d;
```

This is acceptable provided we need this only once or a few times but could be too time consuming for frequent usage like in a loop.

How can one reduce the overhead?

The use of intermediate files for input and output is, evidently, a cost factor. Using the more powerful #pipe instruction we would get rid of these files, e.g. by the following simplified FORM program:

```
#define cmd . . .
Symbol d;
Local withGCD =(2*d^4+3*d^3-22*d^2-13*d+30)/(d^3-11*d+10);
.sort
$EXP=withGCD;
.sort
Local noGCD =
#pipe echo "‘$EXP’" | ‘cmd’
          ;
Print;
.end
```

(for the full working example see Appendix B.2). Here instead of saving the input expression to a file we put it into the command line[3] and read the results back directly from the external command output.

But the overhead is not much less: for each evaluation we have to start a new copy of the external program. This is because the pipe establishes only a unidirectional channel with the external program, so we are not able to organize a dialog with it.

In principle, the #pipe preprocessor instruction can be used in order to establish a real dialog of FORM with the external program, see Appendix C. But this is quite tricky, extremely inconvenient and hardly applicable in practice. In order to extend the FORM functionality we need a really full duplex channel to the external program.

3.3 Dialog with the external program

Recently a full duplex interface has been implemented. The new preprocessor instruction #external starts the system command opening an input-output channel for it. After the instruction #external returns the control to the FORM program, external programs initiated by the instruction continue to run. The standard input and output are intercepted by FORM and not connected with any terminal device.

[3] Of course, the same trick we could use with the \#system instruction; note however that this way the length of the input expression is restricted to a rather small value. This is more an illustration of the concept rather than a working approach.

Generally speaking, the system command initiates several processes starting several external programs combined into the "job". We refer to the job initiated by the instruction `#external` as the "external command".

The instruction `#toexternal` is used to send some text to the running external command. Its syntax is similar to the `#write` instruction.

The instruction `#fromexternal` is used to read the text from the running external command. Its syntax is similar to the `#include` instruction.

The (simplified, for the full example see Appendix B.3) variant of the above FORM program based on this new mechanism may look like:

```
#define cmd . . .
Symbol d;
#external `cmd'

Local withGCD =(2*d^4+3*d^3-22*d^2-13*d+30)/(d^3-11*d+10);
.sort
#toexternal "%E\n",withGCD
Local noGCD =
#fromexternal
        ;
Print;
.end
```

Note that now we have a real dialog.

After the instruction `` #external `cmd' `` the control returns to FORM while all the processes initiated by `` `cmd' `` continue to run. The external command that is running is embedded in the FORM system and visible from the FORM program as a duplex channel. The instruction `#toexternal` provides the external command with input data while the instruction `#fromexternal` reads its resulting output.

It is interesting to compare the speed of all three programs. To minimize irrelevant overhead we repeat the corresponding evaluation 1000 times via a preprocessor `#do` loop and measure the wall-clock time. The results are as follows[4]:

`#system` - based	506.8	seconds
`#pipe` - based	397.6	seconds
`#external` - based	0.8	seconds

[4] The execution times refer to AMD Athlon XP 1800+ computer.

4 Dialog with the external command: detailed description

4.1 General remarks and short introduction

The design of the full duplex channel is not so trivial; automatic bidirectional interaction can be dangerous due to possible deadlocks. This requires careful design and flexibility for all the communication primitives.

There are several modes of starting and finishing the external program. By default, the command is run in a subshell in a new session and in a new process group. Either when FORM finishes or when the `#rmexternal` preprocessor instruction is used, the KILL signal is sent to the whole group. The preprocessor instruction `#setexternalattr` could be used in order to change this default behavior.

Performing the `#fromexternal` instruction FORM continues to read the output of the running external program until the external program outputs a *prompt*. The prompt is a line consisting of a given prompt string. By default, this is an empty string. The prompt string can be changed by means of the preprocessor instruction `#prompt`.

Several external commands can be run simultaneously. The external command that is started last becomes the *current* (active) one. Instructions `#toexternal` and `#fromexternal` deal with the current external command. At any time the current external command can be changed by the instruction `#setexternal`.

This approach was chosen in order to reduce possible overhead. Alternatively, one could implement an additional argument to all primitives containing information about an identifier for each external command.

Special external channels ("pre-opened") could be available provided FORM was started by a parent process with some special command line option or environment variable. In that case FORM is able to communicate with the parent process by means of the same mechanism. This permits FORM to be embedded in other applications, see Appendix D.

An external command can be terminated by the instruction `#rmexternal`.

4.2 Exact syntax

`#external ["prevar"]` systemcommand
starts the command in the background, connecting to its standard input and

output. By default, the external command has no controlling terminal, the standard error stream is redirected to `/dev/null` and the command is run in a subshell in a new session and in a new process group (see the preprocessor instruction `#setexternalattr`).

The optional parameter "prevar" is the name of a preprocessor variable placed between double quotes. If it is present, the "descriptor" (small positive integer number) of the external command is stored into this variable and can be used for references to this external command (if there is more than one external command running simultaneously).

The external command that is started last becomes the "current" (active) external command. All further instructions `#fromexternal` and `#toexternal` deal with the current external command.

`#toexternal "formatstring" [,variables]`
sends the output to the current external command. The semantics of the `"formatstring"` and the `[,variables]` is the same as for the `#write` instruction, except for the trailing end-of-line symbol. In contrast to the `#write` instruction, the `#toexternal` instruction does not append any new line symbol to the end of its output.

`#fromexternal[+|-] ["[$]varname",[maxlength]]`
appends the output of the current external command to the FORM program. The semantics differ depending on the optional arguments. After the external command sends the prompt, FORM will continue with a next line after the line containing the `#fromexternal` instruction. The prompt string is not appended. The optional + or - sign after the name has influence on the listing of the content.

`#fromexternal[+|-]`
The semantics is similar to the `#include` instruction but folders are not supported.

`#fromexternal[+|-] "[$]varname"`
is used to read the text from the running external command into the preprocessor variable `varname`, or into the dollar variable `$varname` if the name of the variable starts with the dollar sign "$".

`#fromexternal[+|-] "[$]varname" maxlength`
is used to read the text from the running external command into the preprocessor (or dollar) variable `varname`. Only the first `maxlength` characters are stored.

The prompt is a line consisting of a single prompt string. By default, this is an empty string. The prompt can be changed by means of the instruction

`#prompt`:

`#prompt [newprompt]`
Sets a new prompt for the current external command (if present) and all further (newly started) external commands.

If `newprompt` is an empty string, the default prompt (an empty line) will be used.

`#setexternal n`
sets the "current" external command. The instructions `#toexternal` and `#fromexternal` deal with the current external command. The integer number `n` must be the descriptor of a running external command.

`#rmexternal [n]`
terminates an external command. The integer number `n` must be either the descriptor of a running external command, or 0.

If `n` is 0, then all external programs will be terminated.

If `n` is not specified, the current external command will be terminated.

The action of this instruction depends on the attributes of the external channel (see the preprocessor instruction `#setexternalattr`). By default, the instruction closes the commands' IO channels, sends a KILL signal to every process in its process group and waits for the external command to be finished.

`#setexternalattr` list_of_attributes
sets attributes for *newly started* external commands. Already running external commands are not affected. The list of attributes is a comma separated list of pairs attribute=value, e.g.:

```
#setexternalattr shell=noshell,kill=9,killall=false
```

Possible attributes are:

- `kill` specifies the signal to be sent to the external command either before the termination of the FORM program or by the preprocessor instruction `#rmexternal`. By default this is 9 (SIGKILL). Number 0 means that no signal will be sent.
- `killall` Indicates whether the kill signal will be sent to the whole group or only to the initial process. Possible values are "`true`" and "`false`". By default, the kill signal will be sent to the whole group.
- `daemon` Indicates whether the command should be "daemonized", i.e. the initial process will be passed to the init process and will belong to the new process group in the new session. Possible values are "`true`" and "`false`".

By default, “true”.

- shell specifies which shell is used to run a command[5]. By default this is “/bin/sh -c”. If set shell=noshell, the command will be stared by the instruction #external directly but not in a subshell, so the command should be a name of the executable file rather than a system command. The instruction #external will duplicate the actions of the shell in searching for an executable file if the specified file name does not contain a slash (/) character. The search path is the path specified in the environment by the PATH variable. If this variable isn’t specified, the default path “:/bin:/usr/bin” is used.
- stderr specifies a file to redirect the standard error stream to. By default it is “/dev/null”. If set stderr=terminal, no redirection occurs.

Only attributes that are explicitly mentioned are changed, all others remain unchanged. Note, changing attributes should be done with care. For example,

```
#setexternalattr daemon=false
```

starts a command in the subshell within the current process group with default attributes kill=9 and killall=true. The instruction #rmexternal sends the KILL signal to the whole group, which means that also FORM itself will be killed.

The example:

```
symbol a,b;

#external "n1" cat -u

#external "n2" cat -u

*  cat simply repeats its input. The default prompt is an
*  empty line. So we use "\n\n" here -- one "\n" is to finish
*  the line, and the next "\n" is the prompt:
#toexternal "(a+b)^2\n\n"

#setexternal ‘n1’
*  For this channel the prompt will be "READY\n":
#toexternal "(a+b)^3\nREADY\n"

#setexternal ‘n2’
*  Set the default prompt:
```

[5] Starting an external command in a subshell permits to start not only executable files but also scripts and pipelined jobs. The disadvantage is that there is no way to detect failure upon startup since usually the shell is started successfully.

```
18  #prompt
19  Local aPLUSbTO2=
20  #fromexternal
21          ;
22
23  #setexternal ‘n1’
24  #prompt READY
25  Local aPLUSbTO3=
26  #fromexternal
27          ;
28
29  #rmexternal ‘n1’
30  #rmexternal ‘n2’
31
32  Print;
33  .end
```

Two external channels are opened in lines 3 and 5. The UNIX utility “`cat`” simply repeats its input. The option “`-u`” is used to prevent the output buffering. The option is ignored by the GNU `cat` utility but is mandatory for non-GNU versions of `cat`.

After line 5 the current external channel is ‘`n2`’. The default prompt is an empty line so in line 10 “`\n\n`” is used – one “`\n`” is to finish the line, and the next “`\n`” is the prompt.

Line 12 switches the current channel to ‘`n1`’. For this channel the prompt will be “`READY`”, see line 24, hence the expression is finished by “`\nREADY\n`”.

Line 16 switches to the ‘`n2`’ external channel and line 18 sets the default prompt (which is extra in this example since the default prompt was not changed up to now).

Results (just a literal repetition of the sent expressions) are read in lines 20 and 26.

Lines 29 and 30 close the external channels.

5 Conclusion

An interface like the one described above can be used in order to extend FORM in the spirit of a component model where the term “component model” means that a software system is built from “components”. Components are high level aggregations of smaller software pieces and provide a "black box" building block approach to software construction (see, e.g., [28]).

The communication is performed only via standard input and output streams without any special protocol, i.e., the input is expected to follow the FORM syntax conventions. But this is not restrictive since the communication could easily be established through an intermediate gateway program (also called filter) which performs a translation between the external programs and FORM (see the discussion in Appendix A and E).

There are two important points:

- The interface is implemented at the preprocessor level. This means that the structure of the FORM algebraic processor is not touched and all the FORM advantages persist.
- It would also allow us to implement a model of parallelization, in which several external components are running simultaneously at the preprocessor level while the FORM algebraic processor continues to work on some huge expression.

The technique described is applied directly to FORM swallowing the external program, or to embedding FORM in other applications. However, this mechanism is quite general and can be used also in order to build FORM in the client-server architecture.

The idea is that FORM can use the "`#external...`" instruction in order to start a client or a server. The client (or the server) then establishes a connection with an external application and uses its standard input and output in order to communicate with FORM.

The corresponding library ("formlink") is under development and will be available from the FORM distribution site [2] in the near future.

Acknowledgements

We want to thank K. Chetyrkin, H.-M. Staudenmaier and M. Steinhauser for interesting discussions. We would also like to thank A. Grozin for advises and useful remarks concerning REDUCE and C. Sturm for continuous testing of the software in a real application. The work of M.T. has been supported by the Deutsche Forschungsgemeinschaft in Sonderforschungsbereich/Transregio 9 "Computational Particle Physics". The work of J.V. has been part of the research program of the Dutch Foundation for Fundamental Research of Matter (FOM).

Appendix

A Choosing an external CAS

The technique we presented was developed and tested in an experimental FORM version used in multiloop calculations [29] applying the Laporta algorithm [30]. The implementation requires some operations on polynomials with integer/rational coefficients.

There are surprisingly few decent programs available for this. It should be a program performing some basic operations on polynomials, and it should be completely stateless, i.e. allow to handle input expressions independent of each other. Ideally, this program should be:

(1) fast enough;
(2) able to handle very large expressions;
(3) available on most hardware/software platforms;
(4) open source.

For the above example we found only two suitable programs, REDUCE [31] and FERMAT [32].

REDUCE is fast enough, it is an open source system (but *not* a free one!) but on 32-bit platforms it is rather restrictive, and there is no IA64 (Intel Itanium) - based version of REDUCE at the moment. Appendix E contains an example of a fully functional embedding of REDUCE in FORM.

In our examples [6] we selected FERMAT, "conjugated" with the `#external` -based FORM interface by a gateway program which performs a syntax translation and proper masking [33] in order to isolate FORM from long expressions treated by FERMAT.

The gateway starts FERMAT, reads input from stdin, passes expressions in the variable d (indicating the dimension of space-time in our example), which FORM collected into the function acc, to FERMAT, reads an answer from FERMAT, stores the answer into an internal table, contracts it to the string "`dd(#)`", where "`#`" is the order number of the expression produced by FERMAT, and outputs the result to the stdout. Here is a very short introduction to the ideas behind it. For example, the expression

```
some string + acc((d+1)/(d-1)+d) + another string
```

[6] These examples are based on ref. [29]

will be converted to

```
some string + dd(1) + another string
```

where `dd(1)` is equal to `(d^2+1)/(d-1)`. It is stored internally and may be extracted by means of the command @v, see below. All lines started with the character @ are commands for the gateway. We show some of them (just a few ones):

- @f0 - do not filter the content of acc() through FERMAT;
  @f1 - filter the content of acc() through FERMAT (default);
- @e0 - reduce all rational functions in d in acc() to dd(#) (default); @e1 - expand all dd(#) to corresponding rational functions (not only in acc()!);
- @vsome text @(#) another text - repeat the line after @v, substituting @(#) by the content of the corresponding dd-variable. Example: "@vid dd(1) = @(1);" will result in "id dd(1) = (d^2+1)/(d-1);"
- @sfilename - all stored variables will be saved into the file "filename", one variable per line, in the order as they were stored (i.e., dd(5) will occupy the line number 5);
  @rfilename - all lines from the file "filename" (without trailing '`\n`') will be loaded into the internal table.

FERMAT is not an open source program; it is available only for Macintosh, Sun SPARC, and Windows and Linux IA32-based platforms (and even not for all of them) but it is able to handle very large expressions even on 32-bit platforms. FERMAT is approx. 20 times faster than the Mathematica function "Together[]" and a little bit faster than REDUCE. It is used for performing the operations on polynomials arising in coefficient functions; we need only basic polynomial arithmetic (which, in principle, FORM provides itself, but still in an experimental and far from optimized way) and the only non-local operation we really need from FERMAT is a GCD contraction.

FERMAT is an interpreter providing a complete programming language. Most of its commands begin with the symbol `&` and have the syntax `&<symbol>` or `&(<symbol>=<value>)`, as in `&q` for quitting the program. There are also some other classes of commands and the terminal I/O commands ! and ?. Entered commands are separated by a new line or by a semicolon.

Entered arithmetic expressions are reduced to some unique form and printed to the terminal. Two arithmetic modes are possible in FERMAT, rational arithmetic and modular. A session with FERMAT starts in rational mode (see example below). All arithmetic is that of rational numbers and the GCD of a numerator and a denominator is contracted. The user may change the ground field by converting it into a polynomial ring entering the "adjoin polynomial" command, `&J`. After that, the user may enter any polynomial sums and ratios and FERMAT will bring them into a unique form of a rational function (“quolynomial”).

In the following example of the FERMAT session the character “>” is the FERMAT prompt, characters after the prompt are typed by the user, and strings without the prompt are printed by FERMAT. We do not show some of the housekeeping information printed by FERMAT:

```
1  >21/14
2   3 / 2;  or  1.5000000000000
3  >&d
4   Enter display size.
5  >0
6  >21/14
7   3 / 2
8  >&(J=d)
9  >(d^2 - 4)/(d+2)
10   d - 2
11 >&(J=x)
12 >(x^2 + 2*x*d +d^2)^2/(x+d)
13   x^3 + (3d)x^2 + (3d^2)x + d^3
```

The command `&d` is used in order to change the number of blocks of 8 significant digits to the right of the decimal point displayed by the interpreter when a non-integer is displayed. If it is 0, this feature is disabled.

In line 8 we change the ground field (rationals) to the ring of polynomials in d, and after line number 11 the ground ring becomes the ring of polynomials in the two variables d and x.

We describe briefly some other FERMAT commands which are used in the rest of the paper.

`&(M=' ')` Change the prompt to just an empty line.

`&(t=0)` Switch off the output of information about the time.

`&U` Toggle switch to display long integers and polynomials in the style of other computer algebra systems (Maple).

B Examples of running external programs

Here we compare all techniques that were available in FORM3 with the new techniques available in FORM 3.2. We provide examples of FORM programs but these programs are only simplified conceptual (but working) examples of how this could be done. General comments:

- As an external CAS, in Appendix B and C we use the program FERMAT, see Appendix A. We assume that the executable file is available from the current directory as "./fermat".
- For simplicity, we do not pay any attention to the line lengths. In reality, some intermediate gateway program (script) should be used in order to make a proper formatting. All the FORM programs in the examples below are started with the statement "`format 254;`". This means, the examples may not work if the total transferred length exceeds 254 characters.
- In order to compare the speed of the examples in all the versions, we repeat the evaluation 1000 times by means of a preprocessor `#do` loop. However, in the example in Appendix C the loop is performed by a "while" loop in the shell.
- The common structure of all the examples is the same. We create the local expression "withGCD" which is Expr. (1), and subsequently repeat 1000 times the process of filtering the expression through FERMAT and putting the resulting Expr. (2) back into FORM.

B.1 Based on the `#system` *instruction*

The following FORM program exploits the `#system` preprocessor instruction to start FERMAT. The information is transferred by conventional files.

```
Off statistics;
Format 254;
#define cmd "./fermat |grep LE=|sed s/^.\*LE=//"
#define prolog "&U; &(J=d); !(\'LE=\',"
#define epilog "); &q"
Symbol d;
Local withGCD = (2*d^4+3*d^3-22*d^2-13*d+30)/(d^3-11*d+10);
.sort
#do i = 1,1000
  #write <finput> "`prolog' %E `epilog'",withGCD
  #system cat finput | `cmd' > foutput
  #remove <finput>
  Local noGCD =
  #include foutput
           ;
  Print;
  .sort
  Drop noGCD;
```

```
19      .sort
20    #enddo
21    .end
```

First the expression `withGCD` is saved to the file “finput” (line 10), then FERMAT is started (line 11). The contents of the input file is passed to the FERMAT standard input and the answer is redirected to the file “foutput”. Note that the file `finput` must be removed (line 12) or else in each iteration the `#write` instruction will append the input expression to the file.

After FERMAT has been started, the content of the file “finput” is sent to it:

```
&U; &(J=d); !('LE=',(2*d^4+3*d^3-22*d^2-13*d+30)/
                                          (d^3-11*d+10)); &q
```

FERMAT prints a lot of text, and one line with the answer:

```
LE=  2*d+3
```

The answer is selected by the filter

```
grep LE=|sed s/^.\*LE=//
```

and redirected to the file “foutput”.

The program is placed into the file “system.frm” and the following command is executed:

```
( time form system.frm ) > system.res 2>&1
```

The result is redirected to the file system.res. Here are several lines from the end of the file system.res:

```
   noGCD =
      3 + 2*d;

     .end

real  8m26.815s
user  5m4.010s
sys   0m24.250s
```

B.2 Based on the `#pipe` instruction

The following FORM program uses the `#pipe` preprocessor instruction to start FERMAT:

```
1  Off statistics;
2  #define cmd "|./fermat |grep LE=|sed s/^.\*LE=//"
3  #define prolog "\&U\; \&\(J=d\)\; \!\(\'LE=\',"
4  #define  epilog "\)\; \&q"
5  Symbol d;
6  Local withGCD =(2*d^4+3*d^3-22*d^2-13*d+30)/(d^3-11*d+10);
7  .sort
8  $EXP=withGCD;
9  .sort
10 #do i=1,1000
11   Local noGCD =
12   #pipe echo `prolog' "`$EXP'" `epilog' `cmd'
13            ;
14   Print;
15   .sort
16   Drop noGCD;
17   .sort
18 #enddo
19 .end
```

The initial expression is passed to FERMAT after the initialization commands which use the command "echo" directly from the command line. The answer is read from the FERMAT standard output intercepted by FORM in the instruction `#pipe`.

The local expression `withGCD` defined on line 6 is passed to the preprocessor on line 12 by means of a dollar variable `$EXP` initialized on line 8.

The program is placed into the file "pipe.frm" and the following command is executed:

```
( time form pipe.frm ) > pipe.res 2>&1
```

The result is redirected to the file pipe.res. We show several lines at the end of the file pipe.res:

```
   noGCD =
      3 + 2*d;

    .end
```

```
real  6m37.593s
user  5m3.360s
sys   0m23.750s
```

B.3 Based on the `#external` *instruction*

The following FORM program uses the `#external` preprocessor instruction to start FERMAT:

```
1  Off statistics;
2  Format 250;
3  #define cmd "./fermat"
4  #define init "&d\n0\n&(M=\' \')\n&(t=0)\n&U\n&(J=d)\n"
5  Symbol d;
6  #external `cmd'
7  #toexternal "`init'"
8  #do i = 1,19
9    #fromexternal "tmp"
10 #enddo
11 Local withGCD =(2*d^4+3*d^3-22*d^2-13*d+30)/(d^3-11*d+10);
12 .sort
13 #do i=1,1000
14   #toexternal "%E\n",withGCD
15   #fromexternal "tmp"
16   Local noGCD =
17   #fromexternal
18            ;
19   Print;
20   .sort
21   Drop noGCD;
22   .sort
23 #enddo
24 .end
```

After FERMAT has been started on line 6 the initialization string is sent in line 7. FERMAT answers with 19 lines of text which are read into the preprocessor variable “`tmp`” and subsequently ignored (lines 8-10). After the expression has been sent (line 14), FERMAT produces an empty line (which is read in line 15 and ignored) and the output expression is assigned to the local expression “`noGCD`” (lines 16-18).

The program is placed in the file "extern.frm" and the following command is executed:

```
( time form extern.frm ) > extern.res 2>&1
```

The result is redirected to the file extern.res, of which we show several lines at the end:

```
    noGCD =
       3 + 2*d;

     .end

real  0m0.799s
user  0m0.160s
sys   0m0.010s
```

C Duplex channel with the `#pipe` instruction

A "real" dialog between FORM and some external program is possible only if the full duplex channel is established between FORM and the external program. In principle, this could be done based on the `#pipe` instruction using the following trick (we don't claim this to be the only way in which it can be done).

The "main" processor program could be not FORM but something started by FORM via the `#pipe` instruction. FORM is only waiting for commands from the main processor, performs these commands and returns results back by means of some sockets or named pipes, see Fig. C.1. The FORM program itself is rather trivial:

```
Off statistics;
Format 250;
#pipe ./processor.sh
.end
```

The main functionality is contained in the shell script "processor.sh" [7]:

```
#!/bin/bash
rm -f tofermat fromfermat fromform 2> /dev/null
mkfifo tofermat fromfermat fromform
./fermat < tofermat > fromfermat &
```

[7] This example requires bash-2.05b-alpha1 or later.

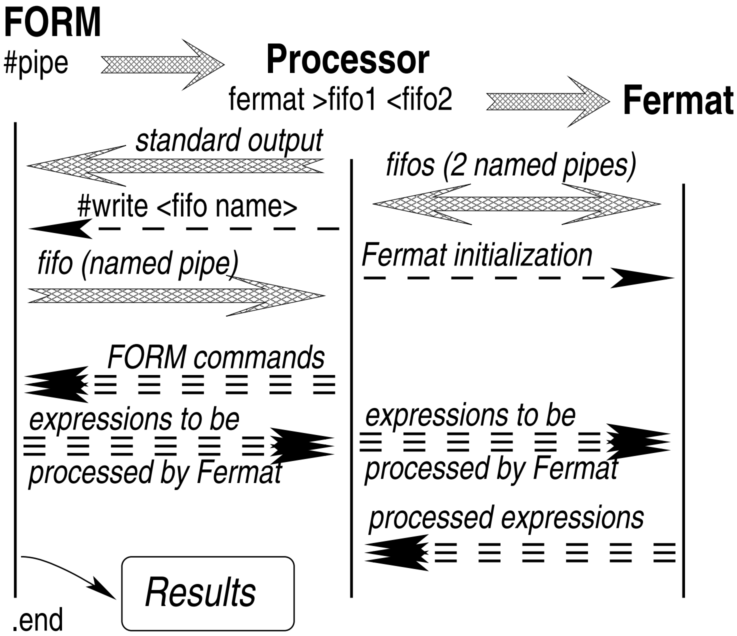

Fig. C.1. The "main" processor program is started by FORM from the trivial program and performs the main control. After initialization, the main program sends a flow of FORM commands to FORM getting back a flow of expressions to be processed by FERMAT. This flow is passed to FERMAT, and the flow of processed expressions is analyzed and transformed back into a flow of expressions transferred to FORM.

```
exec 3> tofermat
exec 4< fromfermat
echo -e "&d\n0\n&(M=' ')\n&(t=0)\n&U\n&(J=d)" >&3
echo "!('PROMPT')" >&3
while [ "$REPLY" != "PROMPT" ] ; do
read -u 4
done
read -u 4; read -u 4
echo '#write <fromform> "PROMPT"'
echo 'Symbol d;'
exec 6< fromform
read -u 6
echo "Local withGCD =(2*d^4+3*d^3-22*d^2-13*d+30)\
                                        /(d^3-11*d+10);"
echo ".sort"
i=0
while [ $i -lt 1000 ] ; do
let i++;
echo '#write <fromform> "%E",withGCD'
echo
read -u 6
```

```
26  echo "$REPLY" >&3
27  read -u 4; read -u 4;
28  echo "Local noGCD = ${REPLY};"
29  read -u 4
30  echo 'Print;'
31  echo '.sort'
32  echo 'Drop noGCD;'
33  echo '.sort'
34  done
35  rm -f tofermat fromfermat fromform
```

This file must be executable, which is achieved by entering the command
"`chmod +x ./processor.sh`"
after the file has been created. The script is the "real" main program and uses FORM for some algebraic evaluations.

The communication between FORM and FERMAT is established by the script using several FIFOs (named pipes). These pipes are created at lines 2 and 3. The named pipe "tofermat" will be used in order to send data to FERMAT while the named pipe "fromfermat" will be used in order to receive data from FERMAT. The named pipe "fromform" will be used in order to get data from FORM while data to FORM are sent by the script via the standard output.

First, the script starts FERMAT (line 4) intercepting the output by opening corresponding FIFOs (lines 5 and 6). Then it initializes FERMAT (line 7) and reads all the unimportant stuff produced by FERMAT (lines 8 – 12).

Then the script asks FORM to create the local expression `withGCD`, reads the result (line 25) and sends it to FERMAT (line 26). The FERMAT answer is, again, passed to FORM (line 28) and then the script asks FORM to print the result (line 30).

The FORM program is placed into the file "wrapper.frm" and the following command is executed:

```
( time form wrapper.frm ) > process.res 2>&1
```

The result is redirected to the file process.res. Here are several lines at the end of the file process.res:

```
   noGCD =
      3 + 2*d;

    drop noGCD;
    .sort
    .end
```

```
real   0m1.340s
user   0m0.500s
sys    0m0.060s
```

Note that logically the main processor program (the script "processor.sh") appears to be dependent on the "stub"-like wrapper FORM program "wrapper.frm". The "main" processor program should really parse the input in order to decide what should be sent to FORM and which data should be passed to FERMAT. Practically, the processor program is a new CAS using FORM to evaluate large-scale expressions, and FERMAT is used for non-local algebraic operations. This is rather similar to the use of Perl scripts in which alternatingly FORM and Maple are used to solve problems in Mathematics [34].

D Embedding FORM in other applications

The external channel instructions permit FORM to swallow an external program. The same mechanism can be used in order to *embed* FORM in other applications.

There is a possibility to start FORM from another program providing one (or more) communication channels (see below). These channels will be visible from a FORM program as "pre-opened" external channels existing after FORM starts. There is no need to open them with the "`#external`" instruction. In this case, the preprocessor variable "`PIPES_`" is defined and is equal to the total number of the pre-opened external channels. Pre-opened external channel descriptors are contained in the preprocessor variables "`PIPE1_`", "`PIPE2_`", etc. For example, if '`PIPES_`' is 3 then there are 3 pre-opened external channels with the descriptors '`PIPE1_`', '`PIPE2_`' and '`PIPE3_`' so e.g. the following instruction could be used:

```
#setexternal `PIPE2_'
```

without

```
#external "PIPE2_"
```

The external channel attributes make no sense for the pre-opened channel (see the preprocessor instruction `#setexternalattr`). Formally, they are as follows:

```
kill=0,
killall=false,
daemon=false,
```

```
stderr=/dev/tty,
shell=noshell
```

In order to activate the pre-opened external channels, the parent application must follow some standards. Here we describe a low-level protocol, the corresponding C-interface is available from the FORM distribution site [2].

Before starting FORM, the parent application must create one or more pairs of pipes [8] . The read-only descriptor of the first pipe in the pair and the write-only descriptor of the second pipe must be passed to FORM as an argument of a command line option "`-pipe`" in ASCII decimal format. The argument of the option is a comma-separated list of pairs "`r#,w#`" where "`r#`" is a read-only descriptor and "`w#`" is a write-only descriptor; alternatively, an environment variable `FORM_PIPES` containing this list can be used [9] . For example, to start FORM with two pre-opened external channels the parent application has to create first four pipes. Lets us suppose the first pipe was created with the descriptors 5 and 6, the second pipe has the descriptors 7 and 8, the third pipe has the descriptors 9 and 10 and the fourth pipe has the descriptors 11 and 12. The descriptors 5 and 8 will be used by FORM as the input and the output for the first pre-opened external channel while the descriptors 9 and 12 will be used by FORM as the input and the output for the second pre-opened external channel.

Then the parent application must start FORM with the following command line option:

```
-pipe 5,8,9,12
```

Upon startup, FORM sends its PID (the Process Identifier) in ASCII decimal format with an appended newline character to the descriptor 8 and then FORM will wait for the answer from the descriptor 5. The answer must be two comma-separated integers in ASCII decimal format followed by a newline character. The first integer corresponds to the FORM PID while the second one is the parent process PID. If the answer is not obtained after some timeout, or if it is not correct (i.e. it is not a list of two integers or the first integer is not the FORM PID) then FORM fails. If everything is correct, FORM creates the pre-opened channel and puts its descriptor in the preprocessor variable "`PIPE1_`".

Then FORM processes the second pair of arguments, "`9,12`".

After all pairs have been processed FORM creates the preprocessor variable

[8] A pipe is a pair of file descriptors, one is for reading and another is for writing. In Linux, see "man 2 pipe".

[9] The command line option overrides the environment veriable.

“PIPES_” and puts into this variable the total number of created pre-opened external channels.

The order of processing the pairs of numbers in the argument is fixed exactly as it was described above i.e. from the left to the right.

E Using REDUCE from FORM

The presented technique can also be used to embed the REDUCE system in FORM.

Transferring data to REDUCE we could ask FORM to use the output format to be compatible with the REDUCE one and the REDUCE output is essentially understandable by FORM although some minor syntax translation should be done. Hence we do need a gateway. The following simple shell script could be used as a gateway:

```
1  #!/bin/sh
2  reduce=/export/pc/reduce
3  MACHINE=linux
4  lisp=psl
5  IMAGE=$reduce/lisp/$lisp/$MACHINE/red/reduce.img
6  export MACHINE
7  export reduce
8  export lisp
9
10 echo 'off int$' > reduceinit
11 echo 'lisp setq(promptstring!*,"")$' >> reduceinit
12 echo 'lisp setq(promptexp!*'",'! )$" >> reduceinit
13 echo 'off nat$'  >> reduceinit
14 echo 'on gcd$'   >> reduceinit
15 echo 'on ezgcd$' >> reduceinit
16 echo '1;' >> reduceinit
17
18 REDUCE="$reduce/lisp/psl/$MACHINE/psl/bpsl -td 128000000 \
19                                                  -f $IMAGE"
20 OUTF1='sed -u  s/\$/\nP\n/g'
21 OUTF2='sed -u /^$/d'
22 cat -u reduceinit -|$REDUCE|$OUTF1|$OUTF2
23 rm -f reduceinit
```

The shell variable `reduce` defined on line 2 is the REDUCE root (dependent of the REDUCE installation!). The variables `MACHINE`, `reduce` and `lisp` should be exported in order for REDUCE to work properly (lines 3-8).

Next the script prepares the initialization commands. These commands are placed into the file `reduceinput` (lines 10-16) which will be deleted afterwards (line 23). The contents of the file are copied to REDUCE using the standard input, see the command `cat` (line 22). The argument "`-`" means that the standard input will be appended (*after* the contents of the initialization file `reduceinput`). The option "`-u`" is ignored by the GNU `cat` utility but is mandatory for non-GNU versions of `cat`.

The problem here is that most utilities attached to something different than the terminal perform so called "block buffering", i.e., the data are passed to the standard output only when the internal buffer is full or when the end-of-file condition is reached. Evidently, we need a different behavior: the data should be passed every time the string is ready. This is so called "line buffering", or just unbuffered output. There are two UNIX utilities supporting this mode, namely `cat` and `sed`. The mode should be switched on explicitly by the `-u` option (the GNU version of `cat` does this by default, and the option `-u` is just ignored).

The prompt is set to "`P`". The end of the REDUCE output expression is the character "`$`", which the output filter "`OUTF1`" (line 20) replaces by a line that contains only the string "`P`". The "`OUTF2`" filter on line 21 removes empty lines. The initialization commands are:
`off int$` – do not remember a history;
`lisp setq(promptstring!*"")$, lisp setqpromptexp!*,'! )$` – set the prompt string to be a new line character;
`off nat$` – switch on a "flat" output;
`on gcd$` – contract a greatest common divisor;
`on ezgcd$` – use the EZ GCD algorithm (Extended Zassenhaus GCD, http://www.uni-koeln.de/REDUCE/3.6/doc/reduce/node120.html).
The input `1;` is added in order to get the prompt string "`P`".

Let us suppose the script is placed into the executable file "`runreduce`" available in the current directory. The following FORM program, named "`usereduce.frm`" demonstrates how to embed REDUCE in FORM:

```
Symbol d;
Format Reduce;
#prompt P
#external ./runreduce
#fromexternal "tmp" 1
Local withGCD =(2*d^4+3*d^3-22*d^2-13*d+30)/(d^3-11*d+10);
.sort
#toexternal "%e\n",withGCD
```

```
9 Local noGCD =
10 #fromexternal
11           ;
12 Print;
13 .sort
14 #toexternal "(%E)*(%E)*(%E);\n",withGCD,withGCD,withGCD
15 Local noGCD1 =
16 #fromexternal
17           ;
18 Print;
19 .sort
20 #toexternal "(%E)^10;\n",withGCD
21 Local noGCD2 =
22 #fromexternal
23           ;
24 Print;
25 .end
```

The external channel to REDUCE is created on line 4. REDUCE responds to initialization by several lines which are read on line 5 and ignored.

Manipulating the output format of FORM is sometimes not so convenient. The “reduce” format option is not guaranteed to be perfect. Using a gateway, we could translate the FORM default output into the REDUCE syntax “on the fly”.

Let us suppose, the following script “`runreduce1`” is available in the current directory:

```
#!/bin/sh
reduce=/export/pc/reduce
MACHINE=linux
lisp=psl
IMAGE=$reduce/lisp/$lisp/$MACHINE/red/reduce.img
export MACHINE
export reduce
export lisp

echo 'off int$' > reduceinit
echo 'lisp setq(promptstring!*,"")$' >> reduceinit
echo 'lisp setq(promptexp!*'",'! )$" >> reduceinit
echo 'off nat$'  >> reduceinit
echo 'on gcd$'   >> reduceinit
echo 'on ezgcd$' >> reduceinit
echo '1;' >> reduceinit
```

```
17
18   REDUCE="$reduce/lisp/psl/$MACHINE/psl/bpsl -td 128000000 \
19                                                      -f $IMAGE"
20   INF1='sed -u s/\^-[0123456789]*/&Pr/g'
21   INF2='sed -u s/\-[0123456789]*Pr/(&)/g'
22   INF3='sed -u -e s/Pr//g -e s/\^/**/g'
23   OUTF0='sed -u  s/\*\*/\^/g'
24   OUTF1='sed -u  s/\$/\nP\n/g'
25   OUTF2='sed -u /^$/d'
26   cat -u reduceinit \
27                -|$INF1|$INF2|$INF3|$REDUCE|$OUTF0|$OUTF1|$OUTF2
28   rm -f reduceinit
```

This file must be executable, which it can be made by entering the command
“`chmod +x ./runreduce1`”
after the file has been created.

The script differs from the former script “`runreduce`” by a sequence of input and output filters, making the complete syntax translation.

Input filters are defined in lines 20-22. The problem here is that REDUCE does not understand expressions like `a^-1`; it needs at least `a^(-1)` instead of it. The standard symbol for the power operation in REDUCE is “`**`” so all “`^`” are replaced by “`**`” (though this is not necessary since REDUCE understans the symbol ‘`^`” in the input correctly).

The “`INF1`” filter translates the “bare” negative powers like “`a^-1`” into the form “`a^-1Pr`”, the filter “`INF2`” converts it into an expression like “`a^(-1Pr)`” and the filter “`INF3`” finally replaces the power operation symbol “`^`” by the REDUCE-standard “`**`” and removes the auxiliary tag “`Pr`”.

The output filter “`OUTF0`” (line 23) translates the REDUCE power operation “`**`” to the FORM notation “`^`”. In principle, this is not necessary since FORM understands the REDUCE-like input notation.

The “`usereduce.frm`” should now be modified: line 2 (“`Format Reduce;`”) must be removed, and line 4 must be changed into the following one:

```
#external ./runreduce1
```

Note, REDUCE will produce some strange line breaks when the numbers become too big, hence a simple gateway like `runreduce1` is not completely suitable. If, for instance, line number 20 of the file “`usereduce.frm`” is changed into

```
#toexternal "(%E)^100;\n",withGCD
```

FORM fails reading the data produced by REDUCE.

The problem is that REDUCE could break the line just before the symbol “`*`” and FORM ignores the next line assuming this is a comment. The comment character could be changed but in general it is more reliable to get the answer without linebreaks at all.

Unfortunately, there is no standard UNIX utility permitting unbuffered operations on the end-of-line symbol. The only way here is to write some small program, e.g., the following Perl script placed in the file “`smallfilter.pl`”:

```
1  #!/usr/bin/perl
2  $|=1;
3  while(<>){
4    $_ =~ s/\*\*/^/g;
5    $_ =~ s/\n//g;
6    $_ =~ s/\$/\nP\n/g;
7    print;
8  }
9
```

This file must be executable, which it can be made by entering the command “`chmod +x ./smallfilter.pl`”
after the file has been created.

This simple script transfers the REDUCE power operation expression (`**`) to the FORM one (`^`), adds the prompt to the end of the output expression and glues all lines into a single string.

The command in line 2 switches off buffering.

Line 4 of “`smallfilter.pl`” is equivalent to the filter “`OUTF0`” in line 23 of the script “`runreduce1`”. Line 6 of “`smallfilter.pl`” is equivalent to the filter “`OUTF1`” in line 24 of the script “`runreduce1`”, and line 5 of “`smallfilter.pl`” extends the functionality of the filter “`OUTF2`” in line 25 of the script “`runreduce1`” just gluing a whole portion of the output into a single line.

Using this script, line (26,27) in the file “`runreduce1`” must be written as

```
cat -u reduceinit -|$INF1|$INF2|$INF3|$REDUCE|./smallfilter.pl
```

and the lines 23-25 should be removed. The resulting gateway is usable without any limitations.

Of course, it is much better to write the gateway as a whole program from the beginning. The corresponding program can be written in Perl:

```
1  #!/usr/bin/perl
2  use IPC::Open2;
3  $|=1;
4
5  $reduce="/export/pc/reduce";
6  $MACHINE="linux";
7  $lisp="psl";
8  $IMAGE="$reduce/lisp/$lisp/$MACHINE/red/reduce.img";
9  $ENV{"MACHINE"}=$MACHINE;
10 $ENV{"reduce"}=$reduce;
11 $ENV{"lisp"}=$lisp;
12
13 $cmd="$reduce/lisp/psl/$MACHINE/psl/bpsl";
14 $arg=" -td 128000000 -f $IMAGE";
15 open2(*R, *W, $cmd . $arg);
16
17 print W  "off int\$\n";
18 print W "lisp setq(promptstring!*,\"\")\$\n";
19 print W "lisp setq(promptexp!*,'! )\$\n";
20 print W "off nat\$\n";
21 print W "on gcd\$\n";
22 print W "on ezgcd\$\n";
23 print W "1;\n";
24 $outp=<R> until $outp eq "1\$\n" ;
25
26 while(<>){
27    $_ =~ s/\^\-([0-9]+)/^(-$1)/g;
28    $_ =~ s/\^/**/g;
29    print W $_;
30    $RANSW=m/;/;
31    while($RANSW){
32       $outp=<R>;
33       $outp =~ s/\*\*/^/g;
34       $outp =~ s/\n//g;
35       $outp =~ s/(\$)/\nP\n/g;
36       print $outp;
37       $RANSW=($1 ne '$');
38    }
39 }
40 print W "bye;\n";
41 <R>;<R>;
42 close(R);
43 close(W);
```

REDUCE is started (line 15) by means of a standard IPC::Open2 module (line 2). Line 3 switches off buffering. Lines 5 - 8 set necessary paths [10] and some environment variables which are exported on lines 9 - 11. Initialization commands (lines 17 - 23) are the same as discussed above (lines 10 - 16 of the file "`runreduce`"). In contrast to shell scripts, the REDUCE response is read directly from the script and ignored (line 24).

Lines 26 - 39 contain the main loop.

The script reads line by line from the standard input (in the conditional expression of the `while` loop, line 26) into the default variable `$_`. After some syntax translation (lines 27 and 28) the translated line is sent to REDUCE (line 29). Line 27 corresponds to the input filters `INF1` and `INF2` and line 28 corresponds to the filter `INF3` of the script "`runreduce1`" (see lines 20 - 22 of the file "`runreduce1`"). In principle, the line 28 is not needed since REDUCE understands the power symbol "`^`" in the input expression.

Once the script detects a semicolon in an input line (lines 30, 31) it starts to read an answer from REDUCE, lines 31 - 38, performing a (generally speaking, unnecessary) syntax translation from REDUCE to FORM (line 33). All newline characters are removed (line 34) and the REDUCE end-of-expression character "`$`" is converted to the prompt "`P`". If such a conversion occurs, the variable "`RANSW`" becomes `false` (line 37) and the script continues to read the standard input.

The script should be placed into an executable file "`runreduce.pl`" and can be used with the FORM program "`usereduce.frm`", where line 4 must be replaced by:

```
#external ./runreduce.pl
```

and line number 2 must be removed. Also line 5 must be removed since now the gateway removes all the messages produced by REDUCE at initialization, see line 24.

All these examples are available on-line from the FORM distribution site [2].

References

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

[10] Remember, in order to run the script, the path to the REDUCE root ("`/export/pc/reduce`") should be changed according to the local REDUCE installation.

[2] All programs and examples relating to this publication can be obtained from the directory publications/external in the FORM distribution site http://www.nikhef.nl/~form.

[3] M. Tentyukov J.A.M. Vermaseren and H.M. Staudenmaier, "ParFORM: Recent development", To appear in: Nuclear Instruments and Methods in Physics Research Section A; cs.SC/0510093

[4] S. Moch, J. A. M. Vermaseren and A. Vogt, Nucl. Phys. B **688** (2004) 101, arXiv:hep-ph/0403192;

[5] A. Vogt, S. Moch and J. A. M. Vermaseren, Nucl. Phys. B **691** (2004) 129, arXiv:hep-ph/0404111;

[6] J. Blumlein and J. A. M. Vermaseren, Phys. Lett. B **606** (2005) 130, arXiv:hep-ph/0411111;

[7] P. A. Baikov, K. G. Chetyrkin and J. H. Kuhn, Phys. Rev. Lett. **95** (2005) 012003, arXiv:hep-ph/0412350;

[8] Y. Schröder and A. Vuorinen, JHEP **0506** (2005) 051, arXiv:hep-ph/0503209;

[9] J. A. M. Vermaseren, A. Vogt and S. Moch, Nucl. Phys. B **724** (2005) 3, arXiv:hep-ph/0504242;

[10] R. Bonciani and A. Ferroglia, Phys. Rev. D **72** (2005) 056004 arXiv:hep-ph/0507047;

[11] P. A. Baikov, K. G. Chetyrkin and J. H. Kuhn, Phys. Rev. Lett. **96** (2006) 012003, arXiv:hep-ph/0511063;

[12] Y. Schröder and M. Steinhauser, JHEP **0601** (2006) 051, arXiv:hep-ph/0512058.

[13] K. G. Chetyrkin, J. H. Kuhn and C. Sturm, arXiv:hep-ph/0512060;

[14] T. Aoyama, M. Hayakawa, T. Kinoshita and M. Nio, arXiv:hep-ph/0512288;

[15] S.A. Larin, F.V. Tkachov and J.A.M. Vermaseren, Preprint NIKHEF-H/91-18 (1991).

[16] M. Steinhauser, Comput. Phys. Comm. **134** (2001) 335; hep-ph/0009029

[17] P. Nogueira, J. Comp. Phys. **105** (1993) 279.

[18] R. Harlander, T. Seidensticker and M. Steinhauser, Phys. Lett. B **426** (1998) 125 (hep-ph/9712228); T. Seidensticker, hep-ph/9905298

[19] M. Tentyukov, J. Fleischer, Comput. Phys. Comm. **132** (2000) 124

[20] M. Tentyukov, J. Fleischer, *"Sophisticated Momenta Mapping with DIANA"*, to appear in Nucl. Instr. Meth. Section A; hep-ph/0510414

[21] F. Jegerlehner, O. Tarasov, Nucl.Phys.Proc.Suppl. **116** (2003) 83; hep-ph/0212004

[22] A. Lorca, T. Riemann, Comput.Phys.Commun. **174** (2006) 71; hep-ph/0412047

[23] A.Pukhov *et al*, hep-ph/9908288

[24] V. Bunichev, A. Kryukov and A. Vologdin, Nucl. Instr. Meth. **A502** (2003), 564; V. Bunichev and A. Kryukov, *"FORM in CompHEP"*, to appear in Nucl. Instr. Meth. Section A

[25] F. Yuasa *et al.* Prog. Theor. Phys. Suppl. **138** (2000) 18

[26] http://sanc.jinr.ru/sanc/; A. Andonov *et al*, Comput. Phys. Comm. **174** (2006) 481

[27] T. Hahn and M. Perez-Victoria, Comput. Phys. Commun. **118** (1999) 153 (hep-ph/9807565);
http://www.feynarts.de/formcalc/

[28] C. Szyperski, "Component Software: Beyond Object-Oriented Programming", Addison-Wesley, 2002 (2nd Edition)

[29] K. G. Chetyrkin *et al*, Eur.Phys.J. **C40** (2005) 361-366; K.G. Chetyrkin, J.H. Kühn and C. Sturm, hep-ph/0512060; K.G. Chetyrkin *et al*, hep-ph/0601165.

[30] S. Laporta, Int.J.Mod.Phys. **A15** (2000) 5087; hep-ph/0102033

[31] http://reduce-algebra.com/

[32] http://www.bway.net/~lewis

[33] C. Anastasiou, A. Lazopoulos, JHEP 0407 (2004) 046; hep-ph/0404258

[34] Jan A. Sanders and Jing Ping Wang. "Combining Maple and Form to decide on integrability questions", Comp. Phys. Comm. **115** (1998) 447-459.