Further developments of FORM

T Ueda[1], T Kaneko[2], B Ruijl[3] and J A M Vermaseren[4]

[1] Faculty of Science and Technology, Seikei University, Musashino, Tokyo 180-8633, Japan
[2] KEK, Tsukuba, Ibaraki 305-0801, Japan
[3] Institute for Theoretical Physics, ETH Zürich, 8093 Zürich, Switzerland
[4] Nikhef Theory Group, 1098 XG Amsterdam, The Netherlands

E-mail: `tueda@st.seikei.ac.jp, toshiaki.kaneko@kek.jp, bruijl@phys.ethz.ch, t68@nikhef.nl`

Abstract. FORM is a symbolic manipulation system, which is especially advantageous for handling gigantic expressions with many small terms, as often occurs in real problems in perturbative quantum field theory. In this work we describe some main features of FORM, such as the preprocessor and $-variables with emphasizing on benefit of metaprogramming, and introduce a new feature: a topology generator.

1. Introduction

The theoretical particle physics community has needed complicated and cumbersome computations based on perturbative quantum field theory in order to precisely predict or explain observable quantities that have been measured in experiments. Nowadays the cutting-edge research often requires symbolic calculations that can never be performed by hand and thus it is common to use computer algebra systems. FORM [1, 2, 3, 4] is one of such software programs that has been used for formula manipulation especially in big calculations. Other computer algebra systems that have been used in theoretical particle physics include `SCHOONSHIP` [5, 6, 7], `REDUCE` [8] and `GiNaC` [9] as well as versatile systems like `Mathematica` and `Maple`. FORM is advantageous for handling gigantic mathematical expressions that do not fit inside the physical memory of computers; its dedicated sorting algorithms, which utilize sequential access to hard disk drives, enable users to manipulate huge expressions on disks efficiently. There are also parallel versions of FORM with POSIX Threads [10] and MPI [11], so that users benefit from modern multicore computer architecture.

2. Daily FORM coding

Here we try to provide some pedagogical examples of FORM programs for beginners. In FORM, one of the basic operations is replacing a part of each term by using pattern matching. Typically, users first define expressions to be manipulated and then write code with pattern matchings in order to specify how they want to manipulate their expressions.

Let us look at our first example in Listing 1. The first 2 lines declare objects that we will use in this example: a (commuting) function `fib` and a symbol `n`. Functions in the FORM language are not *functions* in the mathematical sense or those in other program languages; they just mean objects that can have zero, one or more arguments. The third line defines our input expression, which has a term of the `fib` function with a numeric argument, representing

Listing 1. A simple program to compute Fibonacci numbers.

```
1  CFunction fib;
2  Symbol n;
3  Local F = fib(10);  * Find the 10th Fibonacci number.

4  repeat id fib(n?{>=3}) = fib(n - 1) + fib(n - 2);
5  id fib(2) = 1;
6  id fib(1) = 1;

7  Print;
8  .end
```

a Fibonacci number. The goal of this small program is to evaluate Fibonacci numbers in the input. The next 3 lines specify the manipulations to be applied for each term. The symbol n with a question mark followed by a constraint `n?{>=3}` in the left-hand side of the `id` statement at Line 4 is a wildcard for a pattern matching. If a term has a part matched to this, then the corresponding part will be replaced in the right-hand side with an adequate substitution for n, which is the famous recursion relation of Fibonacci numbers. We `repeat` this replacement while all `fib` functions have (integer) arguments greater than or equal to 3. Then `fib(2)` and `fib(1)` are identified with 1 in a such way that all Fibonacci numbers with positive integer arguments are eventually all reduced to numbers. Running FORM with this example prints `F = 55`.

It is well known that the naïve use of the recursion relation of Fibonacci numbers is highly inefficient because of the exponential growth of the number of evaluations. If one gives the input `F = fib(30)` for the above example, then it leads to generating 832040 terms. This still works though it may take more time and FORM starts to print a longer statistics in its output. FORM has a set of sort algorithms for terms in a hierarchical way based on the merge sort, which works relatively well even on disk storage. This is why FORM is advantageous for manipulating extremely huge expressions that cannot fit in physical memory. The longer statistics in the case of `fib(30)` indicates that FORM enters a new stage of sorting.

The first example never works with `F = fib(1000)` because of its huge number of evaluations. Instead, we will see the next program (Listing 2), which utilizes memorization on Fibonacci numbers.

For this purpose, we use two very powerful features of FORM:

- The preprocessor: this processes preprocessor instructions starting with `#` and manipulates the input text before sending it to the compiler. It provides preprocessor variables and their substitutions into the text, conditional branching, loop constructs, procedures (subroutines), and eventually provides a functionality of metaprogramming.
- $-variables: variables starting with `$` to store small expressions (expected to fit in physical memory), which can be accessed both in compile-time (i.e., by the preprocessor) and run-time.

The idea of the program in Listing 2 is as follows. The first half of the program, up to `.sort` at Line 12, searches for the `fib` function from the input expressions, and determines the maximum argument. Once the maximum argument is known and stored into the preprocessor variable `N` at Line 13, then we build a pre-computed table for Fibonacci numbers up to N with an explicit memorization during their calculations. This is performed only once in compile-time. In run-time, all we need is to apply the pre-computed table for all `fib` functions, as at Line 24. If there is no `fib` at all in the input, then the code to build the table and apply it is discarded due to the `#if`...`#endif` construction. In this way, a result of one part of a FORM

Listing 2. An improved program to compute Fibonacci numbers. The preprocessor and $-variables are used to determine the maximum argument of `fib` functions in run-time and efficient evaluation of Fibonacci numbers with explicit memorization in compile-time.

```
CFunction fib;
Symbol n;
* User input: suppose we don't know the maximum value.
Local F = fib(1000);

* Find the maximum argument.
#$nmax = 0;
if (match(fib(n?$n)));
  $nmax = max_($nmax, $n);
endif;
ModuleOption local, $n;
ModuleOption maximum, $nmax;
.sort

#define N "`$nmax'"
#if `N' > 0
* Build a pre-computed table.
  CTable sparse, check, fibtab(1);
  Fill fibtab(1) = 1;
  Fill fibtab(2) = 1;
  #do i = 3, `N'
    #$value = fibtab(`i' -1) + fibtab(`i' - 2);
    Fill fibtab(`i') = `$value';
  #enddo
* And use the table.
  id fib(n?) = fibtab(n);
#endif

Print;
.end
```

program can change the program flow in another part of the program via the preprocessor. Such small optimizations by metaprogramming make a difference when one has to process very big expressions consisting of millions or billions of terms.

3. Recent developments

FORM version 4.2.0, released in July 2017 and presented in the previous ACAT workshop [12], has introduced many new features, the development of which were inspired by realistic physics problems solved by `Forcer` [13] and local R* operations [14] as well as diagram manipulations [15]. We have recently released a minor update, FORM version 4.2.1, in February 2019. It contains several fixes for bugs and performance issues, appeared in running big programs. For example, many annoying crashes related to compressions were fixed.

For detailed instructions for installation of the up-to-date version, please refer to the Wiki page `https://github.com/vermaseren/form/wiki/Installation`.

Listing 3. A program to generate 2-loop self-energy topologies in the ϕ^3 theory.

```
Vectors Q1,...,Q99;
Vectors p1,...,p99;
Set QQ: Q1,...,Q99;  * for external lines
Set pp: p1,...,p99;  * for internal lines
#define NLOOPS "2"
#define NLEGS  "2"
Local F = topologies_(`NLOOPS',`NLEGS',{3,},QQ,pp);
Print +sss;
.end
```

4. Further developments

In computing an amplitude in perturbative quantum field theory, one may encounter problems in graph theory in many ways:

(i) One needs to enumerate and generate all possible Feynman diagrams for an amplitude.

(ii) When one uses libraries to evaluate Feynman integrals, usually one has to bring diagrams to a notation of topologies that is compatible with the libraries. This requires pattern matching for graphs.

(iii) Some tricks in computing Feynman integrals require elaborate implementations of algorithms in graph theory. For example, the local R* operations require determination of ultraviolet and infrared subdivergences originated from subdiagrams.

For the first case, to generate Feynman diagrams, one may use a diagram generator such as `QGRAF` [16] (written in `Fortran`), the graph generator [17] of `GRACE` [18] (written in `C`) and `FeynArts` [19] (written in `Mathematica`). For the second and third cases, however, they occur during symbolic computations for each diagram, thus one may prefer to implement them in a computer algebra system. This tends to be a quite non-trivial and tedious task. Typically computer algebra systems have their own languages as domain-specific languages, specialized for some tasks, which cannot be expected to have expressivity that general-purpose programming languages may have. It is also problematic that the performance of implemented algorithms may not be so excellent in a comparison with implementations in a low-level language like `C`. Hence it would be very convenient and useful if a computer algebra system provides fast and optimized graph routines for users. It seems natural and relatively easy for Form (written in `C` and `C++`) to incorporate the graph generator of `GRACE`.

Although graph libraries available in Form are still at the planning or developing stage, we have already shipped the version 4.2.1 with an experimental function `topologies_` to generate vanilla topologies[1]. The `topologies_` function takes 5 arguments, specifying the number of loops inside generated topologies, the number of external legs, possible degrees of vertices and two sets of vectors representing external lines and internal lines. Listing 3 is an example to generate 2-loop self-energy topologies consisting with 3-point vertices. Listing 4 is its output, which can be visualized as Figure 1. The result is a set of terms and each term represents one topology. Each topology is a product of `node_` functions and each `node_` function stores the vertex ID and a set of coming momenta. Table 1 shows the number of generated topologies and the timings for self-energies with varying the number of loops, up to the 7-loop level.

[1] The `topologies_` function is provided only for an experimental purpose and could be deprecated in future versions.

Listing 4. The output of the program of Listing 3.

```
    F =
        +
           node_(0,-Q1)
          *node_(1,-Q2)
          *node_(2,Q1,-p1,-p2)
          *node_(3,Q2,p1,-p3)
          *node_(4,p2,-p4,-p5)
          *node_(5,p3,p4,p5)
        +
           node_(0,-Q1)
          *node_(1,-Q2)
          *node_(2,Q1,-p1,-p2)
          *node_(3,p1,-p3,-p4)
          *node_(4,p2,p3,-p5)
          *node_(5,Q2,p4,p5)
       ;
```

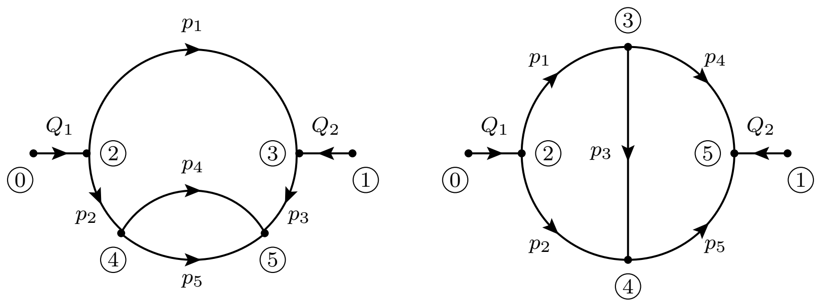

Figure 1. Topologies represented by the output of Listing 4. The circled numbers correspond to the first arguments (vertex IDs) of `node_` functions.

Table 1. Elapsed time for generating topologies. All timings were measured on a Windows laptop.

`NLOOPS`	the number of topologies	Timing
2	2	< 0.01s
3	10	< 0.01s
4	64	< 0.01s
5	519	0.05s
6	4999	0.75s
7	55758	10.12s

5. Summary

As efficient symbolic manipulation is important for the HEP community and others, FORM has evolved with new features as well as bug fixes and improvements. A recent milestone of this evolution was the release of version 4.2.1. We are working on another new feature, graph generations, and others. We hope that these efforts will lead to a release of the next version, shortly. The source code of FORM is hosted on GitHub and available from `https://github.com/vermaseren/form`.

References

[1] Vermaseren J A M 1991 *Symbolic Manipulation with FORM, Tutorial and Reference Manual* (CAN)
[2] Vermaseren J A M 2000 *Preprint* `math-ph/0010025`
[3] Kuipers J, Ueda T, Vermaseren J A M and Vollinga J 2013 *Comput. Phys. Commun.* **184** 1453–67 (*Preprint* `1203.6543`)
[4] Ruijl B, Ueda T and Vermaseren J 2017 *Preprint* `1707.06453`
[5] Veltman M 1967 *CERN Preprint*
[6] Strubbe H 1974 *Comput. Phys. Commun.* **8** 1–30
[7] Veltman M J G and Williams D N 1991 *Preprint* `hep-ph/9306228`
[8] Hearn A C 1968 *Proceedings for the ACM Symposium on Interactive Systems for Experimental Applied Mathematics* pp 79–90
[9] Bauer C W, Frink A and Kreckel R 2000 *J. Symb. Comput.* **33** 1 (*Preprint* `cs/0004015`)
[10] Tentyukov M and Vermaseren J A M 2010 *Comput. Phys. Commun.* **181** 1419–27 (*Preprint* `hep-ph/0702279`)
[11] Tentyukov M, Fliegner D, Frank M, Onischenko A, Retey A, Staudenmaier H M and Vermaseren J A M 2004 *Proceedings for 7th International Workshop on Computer Algebra in Scientific Computing (CASC 2004) St. Petersburg, Russia, July 12-19, 2004* (*Preprint* `cs/0407066`)
[12] Ueda T 2018 *J. Phys. Conf. Ser.* **1085** 022001
[13] Ruijl B, Ueda T and Vermaseren J A M 2017 *Preprint* `1704.06650`
[14] Herzog F and Ruijl B 2017 *JHEP* **05** 037 (*Preprint* `1703.03776`)
[15] Herzog F, Ruijl B, Ueda T, Vermaseren J A M and Vogt A 2016 *PoS* **LL2016** 073 (*Preprint* `1608.01834`)
[16] Nogueira P 1993 *J. Comput. Phys.* **105** 279–89
[17] Kaneko T 1995 *Comput. Phys. Commun.* **92** 127–52 (*Preprint* `hep-th/9408107`)
[18] Yuasa F *et al.* 2000 *Prog. Theor. Phys. Suppl.* **138** 18–23 (*Preprint* `hep-ph/0007053`)
[19] Hahn T 2001 *Comput. Phys. Commun.* **140** 418–31 (*Preprint* `hep-ph/0012260`)