New developments in FORM

Takahiro Ueda

Nikhef Theory Group, Science Park 105, 1098 XG Amsterdam, The Netherlands

E-mail: tueda@nikhef.nl

Abstract. Symbolic computation is an indispensable tool for theoretical particle physics, especially in the context of perturbative quantum field theory. In this work, I will review FORM, one of the computer algebra systems widely used in higher-order calculations, its design principles and advantages. The newly released version 4.2 will also be discussed.

1. Introduction

Recent rapid progress in higher-order calculations in theoretical particle physics would not have been possible without automatization based on symbolic computation software. Indeed, some computer algebra systems arose out of real needs in cumbersome theoretical calculations in perturbative quantum field theory; for example, SCHOONSHIP [1, 2], REDUCE [3, 4], FORM [5, 6, 7] and GiNaC [8].

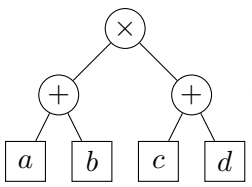
FORM is often referred as the (portable) successor to legendary[1] SCHOONSHIP, which was programmed in assembly language and super fast compared to other programs at that time. Many large-scaled cutting-edge perturbative computations have been performed with FORM since its birth in the late '80s, because of its high performance in processing gigantic expressions, required in this field; nowadays in extreme cases the intermediate expression size can reach a few terabytes.

This work aims to review FORM and give an overview of its design principles. The recently released new version FORM 4.2 will also be briefly discussed.

2. Design principles in FORM

Some computer algebra systems store a mathematical expression in a tree data structure. For example, an expression $(a+b)(c+d)$ can be represented as

$$(a+b)(c+d) \quad \Longrightarrow$$



,

where the operators '×' and '+' are expressed by nodes in the tree and have operands as their children. This is a natural design decision to represent an expression, which has an origin from

[1] Personally I never have had and perhaps never will have the chance to run SCHOONSHIP or access to any legacy computer on which SCHOONSHIP could run.

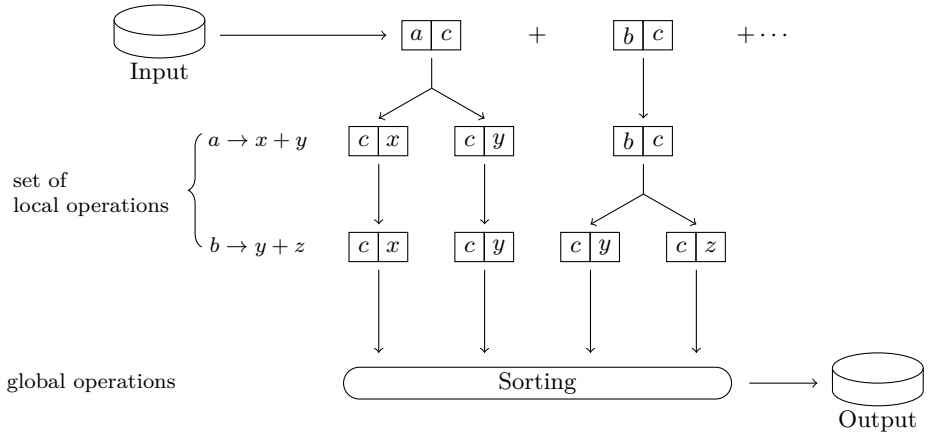


Figure 1. Processing a mathematical expression in FORM.

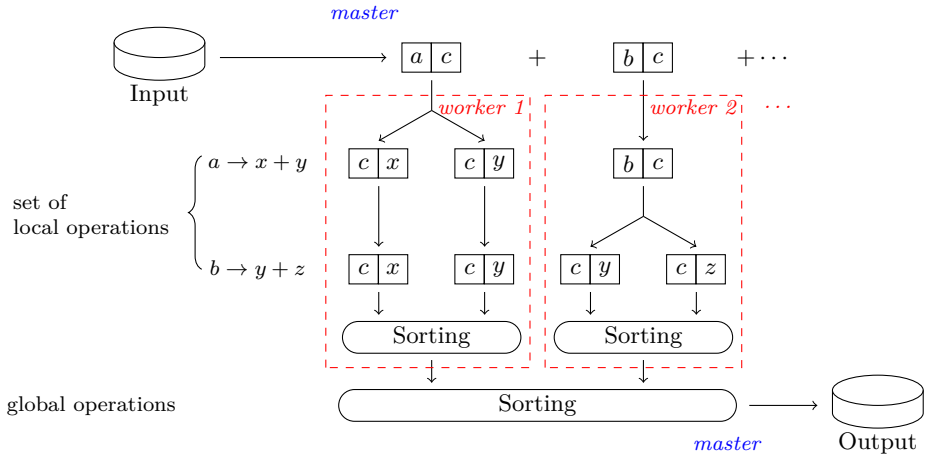


Figure 2. Parallelization of FORM.

LISP-influenced systems. Such a tree structure can keep any input expression in the exact form that the user intended. Symbolic manipulations like simplification may utilize the recursive tree structure and would be elegantly implemented in computer code. However, tree structures without any fixed size commonly require many dynamic memory allocations from the heap. When an expression tree becomes extremely large and does not fit in physical memory, the operating system typically assigns virtual memory on a disk, leading to intolerably slow random access to the disk in every symbolic manipulation.

In FORM, any input expressions are always expanded unless the user intervenes, and then an expression is stored as a sequence of sorted terms:

$$(a+b)(c+d) = ac + ad + bc + bd \quad \Longrightarrow \quad \boxed{a}\boxed{c} \; \boxed{a}\boxed{d} \; \boxed{b}\boxed{c} \; \boxed{b}\boxed{d}\,.$$

The expansion may generate many terms but each term tends to be rather small, expected to easily fit in physical memory or even in CPU cache. Indeed, FORM restricts the size of each term after the expansion to be smaller than some fixed size given by a setup parameter, which enables use of fast memory pools and eliminates many dynamic memory allocations at run-time.

Most of the operations on expressions in FORM are restricted to be 'local' in such a way that they act on each individual term and do not need any reference to other parts of the expression.

Listing 1. Differentiating polynomials.

```
1  Symbol x,n;
2  Local F1 = 1 + x;
3  Local F2 = 2 - 3*x + 5*x^2;
4  id x^n? = n * x^(n-1);
5  Print;
6  .end
```

FORM applies a set of local operations to terms in an expression, term by term, which may generates many terms in general. After that, non-local (or 'global') operations like sorting are performed. This is illustrated in Figure 1. The sorting is based on the merge sort algorithm and only requires sequential access to disks, instead of random access. Furthermore, after the sorting, terms with similar subterms tend to come together; therefore, the incremental encoding works well to compress the total data size, which speeds up the reading and writing. Such an efficient design makes FORM be a suitable choice for handling gigantic expressions exceeding physical memory, in comparison with many of the other computer algebra systems.

Another benefit of the restriction on local operations is that it is relatively straightforward to parallelize processing terms in the master-worker model. The master distributes terms over workers, each worker processes the distributed terms and partially sorts the result, and then the master collects all the results and performs the final sorting (Figure 2). This idea was implemented as ParFORM [9] using MPI and TFORM [10] based on POSIX Threads.

3. Programming in FORM

A good toolkit must provide a good user interface that is suitable and reasonable for the user's purpose. In the FORM programming language, one of the basic operations is term rewriting. The user writes a set of substitutions that will be applied on every term in an imperative programming style, by which one can build algorithms for, e.g., differentiations, integrations and function transformations.

Listing 1 shows a complete example. If FORM is installed on a computer, one can run this example by saving it as a text file, e.g., `example.frm` and executing `form example.frm` from a terminal window. The FORM language is strongly-typed, thus one has to declare symbols or other types of objects before their use. Line 1 declares symbols `x` and `n`. All statements in FORM end with a semicolon '`;`'. Although user-defined objects are case-sensitive, all built-in objects and command names are case-insensitive. Line 2 and 3 define (local) expressions `F1` and `F2`, which will be manipulated in the following code. Line 4 is an executable statement to apply a replacement, which will be acted on every term in the current active expressions, `F1` and `F2` in this case. The command `id` is an abbreviation of `identify`. The question mark '`?`' indicates `n?` is a wildcard, and this statement replaces x^n by nx^{n-1} for any $n \in \mathbb{Z}$, implementing differentiation of polynomials with respect to x. Line 5 instructs FORM to print the expressions at the end of the current *module* (see below). Line 6 is a special instruction to indicate the end of the program.

Note that an `id` statement is allowed to generate more than one term, like `id x = y + z;`, but the left-hand side of an `id` statement cannot have more than one term. For example, the statement `id y + z = x;` is forbidden because it is not a local operation and FORM just prints a compilation error. When an operation generates more than one term, the next operation is applied to each term in a depth-first manner.

Listing 2. Defining a procedure for differentiating polynomials and its call.

```
#procedure derivative(x,m)
  #do i=1,`m'
    id `x'^n? = n * `x'^(n-1);
  #enddo
#endprocedure

#call derivative(y,2)
```

Listing 3. A FORM program consisting of two modules.

```
Symbol a,b,x;
Local F = a*x + x^2;

id x = a + b;

Print;
.sort

if (count(b,1) == 1);  * select terms with b^1
  multiply a/b;
endif;

Print;
.end
```

A remarkable feature of the FORM language is that it embeds a very powerful preprocessor, which can modify the input for the compiler. A preprocessor directive line in FORM starts with a hash mark '`#`', like C-preprocessor directives. Indeed, complicated FORM programs virtually cannot be constructed without using the preprocessor directives. For example, the `#procedure` and `#call` directives are used for procedural programming, in which one makes use of reusable subroutines to construct programs or even bigger subroutines.

Listing 2 is an example of a procedure for differentiating polynomials.[2] The defined procedure can be *called* as `#call derivative(y,2)`, which leads to two sequential `id` statements that give the second derivative with respect to y. Two parameters are passed as preprocessor variables, `x` and `m`, and expanded by backquote/quote pairs inside the procedure. One should realize that this call is done as a text substitution by the preprocessor, hence at compile-time, not at run-time.

A FORM program consists of one or more basic executing blocks that are referred to as modules. A module ends with a special instruction starting with a period '`.`' such as `.sort` and `.end`. When the FORM compiler reaches such an instruction and recognizes the termination of a module, compiled code in the module is executed. All active expressions are processed and then the results are sorted. For example, the program in Listing 3 consists of two modules. First, FORM compiles the first module and executes the code in it. The result for expression `F` is sorted and printed. After that, FORM compiles the second module and executes the code. Note that in this example removing the `.sort` instruction does not change the final result. Inserting a

[2] This procedure is not perfect. It could be optimized such that multiple `id` statements would be combined into one at compile-time, and actually there is a buggy case; `#call derivative(n,1)` gives a result that most likely the user does not want.

Listing 4. A code snippet with compile-time optimization based on run-time information obtained in a previous module.

```
#define HaveX "0"

* Check whether any expressions have x.
if (count(x,1));
  redefine HaveX "1";
endif;

.sort

#if `HaveX'
* This part is skipped at compile-time
* if there is no x in all expressions.
#endif
```

`.sort` instruction may give a speed-up of the program if the sorting gives a much less number of terms by merging or cancellation among the terms, while sorting itself has some overheads. The user has full control of whether or not expressions are sorted at some point by inserting or not inserting a `.sort` instruction.

The fact that a module is compiled after the execution of previous modules means that one can optimize code by the preprocessor at compile-time with run-time information from the previous modules. Listing 4 is an example of compile-time optimization, where a code block is conditionally skipped depending on run-time information obtained in the previous module. Such compile-time optimization techniques make sense when one has to manipulate million of terms at run-time. Tricky and sometimes mystifying metaprogramming with the preprocessor is after all justified by productivity and improvement of the performance of the code.

4. Form 4.2

Currently, the latest stable version of Form is the version 4.2.0 and was released in July 2017 [11]. In comparison with the previous version 4.1, this release contains more than 50 bug fixes and more than 20 new features, including

- generating all matches by `id all`,
- automatic expansion of rational polynomials,
- dictionaries for textual manipulation of output,
- temporary storage for spectator terms,
- stochastic local search for polynomial optimization,
- zero-dimensional sparse tables.

Many of the introductions of new features and improvements in Form have been inspired by use in actual research projects. For this version, the Forcer [12] program and implementing R*-operation on tensor integrals [13] were the main driving force of the development.

Here I would like to focus on a problem that was made solvable in Form 4.2: graph isomorphism by pattern matching on the fly. The standard way of perturbative calculations in quantum field theory begins with generating Feynman diagrams. Feynman diagram generators on the market do the job, but their momentum assignment to diagrams may be different from what the succeeding programs expect. Therefore, one needs to recognize the topology of a

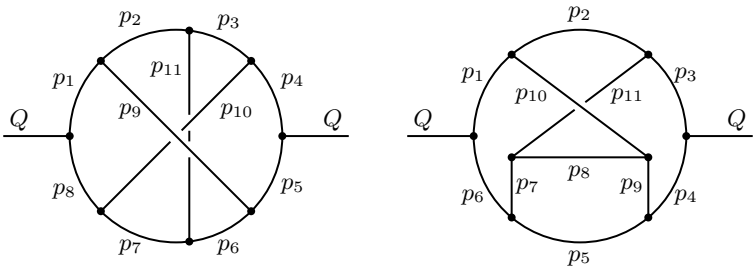


Figure 3. Two isomorphic graphs.

Listing 5. Finding isomorphic mappings.

```
1  CFunction map;              * for isomorphic mapping
2  CFunction vx(symmetric);    * for vertices
3  Vector Q,p1,...,p11,q1,...,q11;

4  Local F1 = vx(Q,p1,p8)*vx(p1,p2,p9)*vx(p2,p3,p11)*
5             vx(p3,p4,p10)*vx(Q,p4,p5)*vx(p5,p6,p9)*
6             vx(p6,p7,p11)*vx(p7,p8,p10);

7  Local F2 = vx(Q,p1,p6)*vx(p1,p2,p10)*vx(p2,p3,p11)*
8             vx(Q,p3,p4)*vx(p4,p5,p9)*vx(p5,p6,p7)*
9             vx(p7,p8,p11)*vx(p8,p9,p10);

10 id vx(Q,q1?,q8?)*vx(q1?,q2?,q9?)*vx(q2?,q3?,q11?)*
11    vx(q3?,q4?,q10?)*vx(Q,q4?,q5?)*vx(q5?,q6?,q9?)*
12    vx(q6?,q7?,q11?)*vx(q7?,q8?,q10?)
13    = map(q1,...,q11);

14 Print;
15 .end
```

generated diagram and reassign the momenta as in the notation of the succeeding programs. More problems related to Feynman diagrams and their solutions are found in [14].

For simplicity, we consider a graph isomorphism problem of *undirected* graphs.[3] The question is stated as follows: Are two graphs in Figure 3 isomorphic after relabeling the edges? If so, what is the isomorphic mapping to obtain one of them from the other?

This problem can be solved in FORM as in Listing 5. Two topologies are expressed as products of a symmetric function, `vx`, whose arguments are vectors representing edges connected to each vertex, and stored in expressions `F1` and `F2`. Lines 10 to 13 perform a pattern matching for the isomorphic test. The pattern is constructed from expression `F1` by replacing the internal edges `p1`, ..., `p11` with wildcards `q1?`, ..., `q11?`. If the pattern matches to `F2`, then the two graphs are isomorphic and we find the isomorphic mapping as a `map` function in the result.[4] One can also obtain all automorphic mappings via generating all matches by `id all` instead of `id`.

[3] The way described here can be extended to the case of *directed* graphs, namely, products of `vx` with *minus vectors* representing momentum directions like `vx(Q,p1,-p8)`. However, the cost of pattern matching with minus vectors is rather high and in practice it is much faster to decompose the problem into pattern matching without directions and then fixing the directions.

[4] In general, the pattern match also occurs when `F2` contains `F1` as its subgraph. Such a case can be easily excluded by checking graph invariants like the number of vertices.

Listing 6. Generation of a pattern at compile-time.

```
1  #$graph1 = F1;
2  #opendictionary wild
3    #do i=1,11
4      #add p`i': "q`i'?"
5    #enddo
6  #closedictionary
7  #usedictionary wild($)
8    id `$graph1' = map(q1,...,q11);
9  #closedictionary
```

Listing 7. Generation of a pattern at run-time.

```
1  $graph1 = F1;
2  id $graph1 * replace_(<p1,q1?>,...,<p11,q11?>)
3    = map(q1,...,q11);
```

Listing 5 works for this example with the specific topology, but is inflexible from the point of view that the explicit pattern is directly written down in the code and it does not work with arbitrary topologies. A more preferable way is to generate a pattern from one of the given expressions on the fly. Two possible solutions are shown in the following. The both uses a *$-variable*, which stores a small expression in physical memory and is accessible both at compile-time and run-time.

The first solution uses dictionaries, a new feature introduced in the version 4.2. Dictionaries store textual mappings that are applied to the output, and originally intended for translation of the output into, for example, the LaTeX format like `mu1` to `\mu_1`. They can be also applied to text expansion of $-variables in the preprocessor, and hence lead to the code in Listing 6. Line 1 stores expression `F1` into a $-variable and line 8 expands it in the preprocessor, but with text replacements defined at line 4. As the result, the same pattern to that in Listing 5 is obtained at compile-time.

Another solution, shown in Listing 7, constructs the pattern at run-time. It makes use of the fact that the left-hand side of an `id` statement can be manipulated at run-time to a certain extent. Of course this causes some overheads at run-time, thus the compile-time solution of Listing 6 is preferable if possible, for example, in the case that `F1` is given in the previous module and one needs to perform the isomorphic test for many graphs with `F1`.

5. Summary

An overview of Form was presented from viewpoints of its design principles and how one can write his or her program in the Form language. The latest version 4.2 and its application to isomorphic test problems were also discussed. Hopefully, users can write more creative and even smarter programs with the newly added features. The source code of Form is hosted on GitHub and available from `https://github.com/vermaseren/form`.

Acknowledgments

I am grateful to my collaborators in the Form project and its physics applications. This work is supported by the ERC Advanced Grant no. 320651, “HEPGAME”.

References

[1] Veltman M 1967 *CERN Preprint*

[2] Strubbe H 1974 *Comput. Phys. Commun.* **8** 1–30

[3] Hearn A C 1968 *Proceedings for the ACM Symposium on Interactive Systems for Experimental Applied Mathematics* pp 79–90

[4] `http://reduce-algebra.sourceforge.net/`

[5] Vermaseren J A M 1991 *Symbolic Manipulation with FORM, Tutorial and Reference Manual* (CAN)

[6] Vermaseren J A M 2000 *Preprint* `math-ph/0010025`

[7] Kuipers J, Ueda T, Vermaseren J A M and Vollinga J 2013 *Comput. Phys. Commun.* **184** 1453–67 (*Preprint* `1203.6543`)

[8] Bauer C W, Frink A and Kreckel R 2000 *J. Symb. Comput.* **33** 1 (*Preprint* `cs/0004015`)

[9] 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`)

[10] Tentyukov M and Vermaseren J A M 2010 *Comput. Phys. Commun.* **181** 1419–27 (*Preprint* `hep-ph/0702279`)

[11] Ruijl B, Ueda T and Vermaseren J 2017 *Preprint* `1707.06453`

[12] Ruijl B, Ueda T and Vermaseren J A M 2017 *Preprint* `1704.06650`

[13] Herzog F and Ruijl B 2017 *JHEP* **05** 037 (*Preprint* `1703.03776`)

[14] Herzog F, Ruijl B, Ueda T, Vermaseren J A M and Vogt A 2016 *PoS* **LL2016** 073 (*Preprint* `1608.01834`)