Introduction to FORM

March 24, 2025

Bakar Chargeishvili

Computer Algebra and Particle Physics - CAPP 2025,
March 24 – 28, 2025

Karlsruhe Institute of Technology

What is FORM?

- A powerful symbolic manipulation system specialized for high-energy physics
- Developement started by Jos Vermaseren at Nikhef
- First released in 1989, with roots in Schoonschip (Martinus Veltman's system from the 1960s)
- Designed to handle extremely large algebraic expressions in quantum field theory
- Distinguishes itself by its speed and memory efficiency for massive calculations
- Used in numerous advanced calculations in particle physics
- Open source since 2000, with ongoing development by the community

Importance of Innovations in Calculation Techniques

- In 1960s Martinus J. G. Veltman developed first CAS - SCHOONSCHIP.
- Later on he was writing:

  ... on the basis of this program [SCHOONSCHIP], the commercially successful software program Mathematica was developed by Wolfram. While most theoreticians were getting lost in all kinds of formulas, I could just get things calculated. That gave me an enormous advantage.

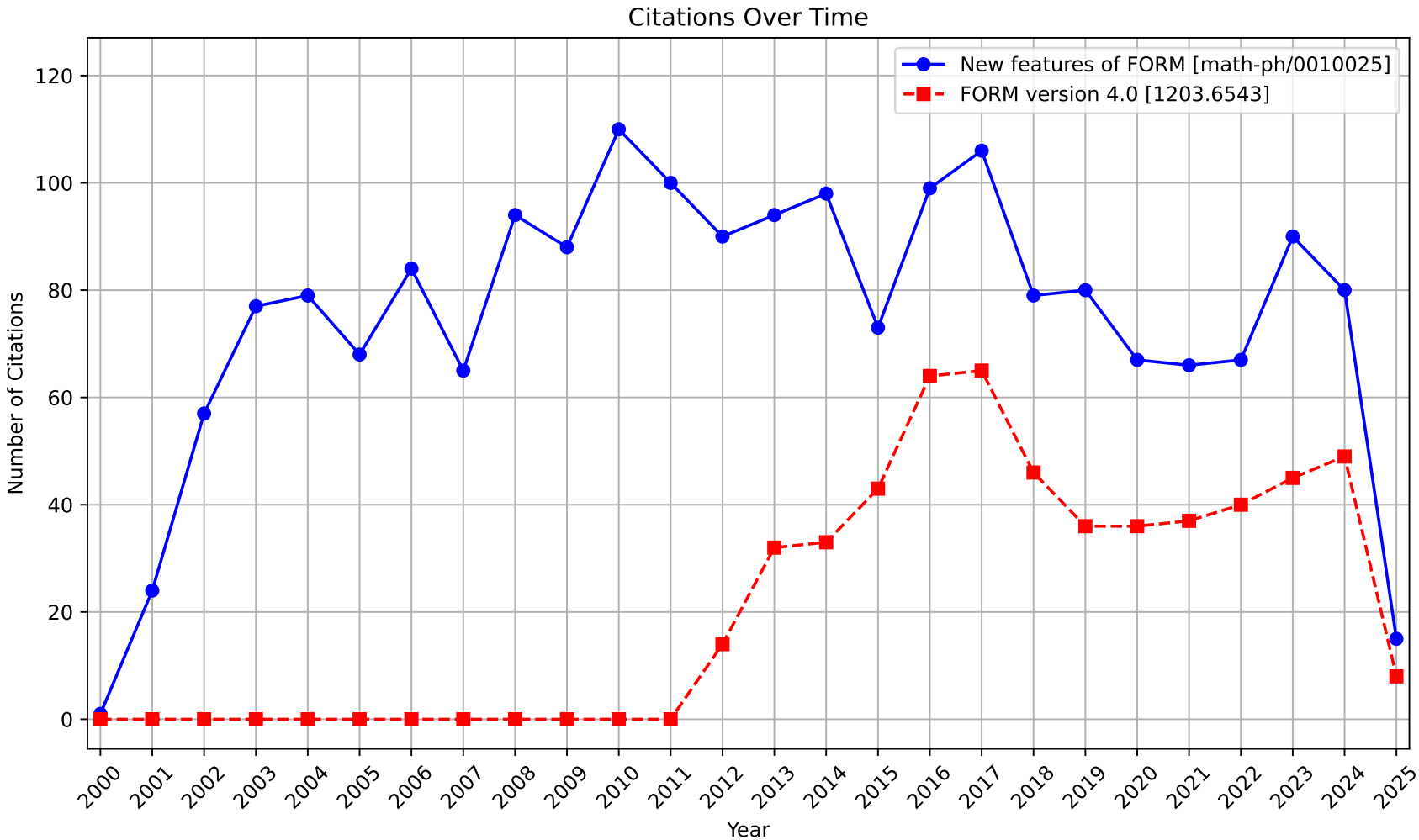
M. Veltman

- Many other computer algebra systems were released afterwards, which are more efficient and user friendly than SCHOONSCHIP.
- Until now the core principle remains unchanged:
  - The user is expected to map the natural language to artifical language constructed by some developer
- But still, not all of CASs are created equally...

Common Beliefs & Misconceptions about FORM

- "FORM is an ancient tool no longer relevant in modern physics"
- "It's not actively maintained and there are a lots of bugs"
- "Modern CAS systems have made FORM obsolete"
- "FORM is not user-friendly with its steep learning curve"
- "It's only useful for specialized calculations that nobody does anymore"
- "The syntax is too cryptic and difficult to learn"
- "It's not compatible with modern computational workflows"
- "Only a handful of experts can actually use it effectively"
- "There's no community support and you are left alone with your problems"

Relevance of FORM



Relevance of FORM Today

- Free and open-source software with an active development community
- High efficiency in handling large polynomial expressions
- Transparent memory management and algorithmic approaches
- Highly portable across different computing environments and platforms
- Minimalistic design focused on performance
- Numerous relevant public libraries are implemented in FORM
- Parallel processing capabilities for modern multi-core systems
- Specialized features designed specifically for quantum field theory
- Continuous improvements with new releases and updates respecting the original design phylosophy

Goals of These Lectures

- Demonstrate core functionality through practical examples
- Assume zero prior knowledge
- Highlight common pitfalls
- Show efficient techniques for large calculations
- Equip you with workarounds for limitations

Today:

- Basic functionalities, basic exercises

Tomorrow:

- Advanced features, advanced exercises

Current state of FORM

- The historical homepage of FORM: https://www.nikhef.nl/~form
- The actual development/discussions: https://github.com/vermaseren/form
- The Reference Manual: https://www.nikhef.nl/~form/maindir/documentation/reference/reference.html
- The latest official release: v4.3.1
- The actual development version: v5.0.0-beta.1
- Installation:
  - From official github repository
  - Using a helper script: https://github.com/magv/hepware

Hello World

hello_world.frm:

```
1   #-
2   #message Hello World
3   .end
```

form hello_world.frm:

```
FORM 5.0.0-beta.1 (Mar  7 2025, v5.0.0-beta.1-122-g638f84b)  Run: Mon Mar 24
    #-
~~~Hello World
  0.00 sec out of 0.00 sec
```

FORM code structure

- The code is organized in *modules*.
- The commonly used modules:
  - .sort
    The general end-of-module. Causes execution of all active expressions, and prepares them for the next module.
  - .end
    Executes all active expressions and terminates the program.
  - .global
    No execution of expressions. It just saves declarations made thus far from being erased by a .store instruction.
  - .store
    Executes all active expressions. Then it writes all active global expressions to an intermediate storage file and removes all other non-global expressions. Removes all memory of declarations except for those that were made before a .global instruction.

The structure of modules

- Modules consist of statements
- Statements should always appear in the following order:

```
Declarations
....
Specifications
....
Definitions
...
Statements
...
Output control statement
```

- FORM manual lists for each keyword which category does it belong to

Example of a valid FORM code


```
#-
* Declarations
Symbols m,n;
* Specifications
Off statistics;
* Definitions
Local A = (m+n)^2;
* Output
Print;
.sort
* Declarations
Symbol z;
* Specifications
Drop;
* Definitions
Local B = A+z;
* Statement
identify n = 3;
* Output
Print;
.end
```

Inner Workings of FORM

Design choice:

- Expressions are always expanded
- Operations inside the module are applied term-by-term

Consequences:

- Possible proliferation of terms to be processed
- Effective memory management
- Difficult to implement anything beyond single term based operations


```
1   Symbols a,b,c;
2   Local F = 14*a-23*b;
3   Print "1: %t";
4   id  a = b+c;
5   Print "2: %t";
6   Print;
7   .end
```

Output:

```
    #-
1:  + 14*a
2:  + 14*b
2:  + 14*c
1:  - 23*b
2:  - 23*b

   F =
      14*c - 9*b;

  0.00 sec out of 0.00 sec
```

To .sort or not to .sort

Compare

```
1  Symbols a,b,c,d,e;
2
3  Local test = (a+b+c+d)^30;
4  id a = -b+5;
5
6  id c = -d+e;
7  Print;
8  .end
```

vs.

```
1  Symbols a,b,c,d,e;
2
3  Local test = (a+b+c+d)^30;
4  id a = -b+5;
5  .sort
6  id c = -d+e;
7  Print;
8  .end
```

- Two fundamental types of functions:

```
1  #-
2  CFunctions A,B,C;
3  Functions  F,G,H;
4
5  L test = G(25)*C(1,2)*A(3,4)*B(6,5)*H(1)*F(4);
6
7  Print;
8  .end
```

Output:

```
test =
     G(25)*H(1)*F(4)*A(3,4)*B(6,5)*C(1,2);

 0.00 sec out of 0.00 sec
```

Substitutions

- Substitutions are made using **identify** (**id**) statement:

```
1  #-
2  CFunction F;
3  Symbols a,b;
4
5  L test = F(1,a);
6  id F(a?,b?) = F(b,a,a+b);
7
8  Print;
9  .end
```

Output:

```
   test =
      F(a,1,1 + a);

  0.00 sec out of 0.00 sec
```

Wildcards

- `a?`, `b?` are wildcards
- They match the objects depending their declaration
- One might have complicated restrictions on wildcards, e.g.`a?{,>-2}`, match symbols which are larger than -2.
- The symbols starting with `?` are ranged wildcards, matching with everything until the end range:

```
1  #-
2  CFunction F;
3
4  L test = F(37,28);
5  id F(?a) = F(1,?a);
6
7  Print;
8  .end
```

Substitutions: Caveat

What are these programs going to produce?

```
1  #-
2  CFunction F;
3  Symbols a,b;
4
5  L test = 1/F(1,a);
6  id F(a?,b?) = F(b,a,a+b);
7
8  Print;
9  .end
```

or

```
1  #-
2  CFunction F;
3  Symbols a,b;
4
5  L test = F(1,a);
6  id a = b;
7
8  Print;
9  .end
```

Argument, EndArgument

Operate on the arguments of a function:

```
1   #-
2   CFunction F;
3   Symbols a,b;
4
5   L test = F(1,a);
6   Argument F;
7       id a = b;
8   EndArgument;
9
10  Print;
11  .end
```

Output:

```
   test = F(1,b);

  0.00 sec out of 0.00 sec
```

Revisiting to .sort or not to .sort example


```
1   Symbols a,b,c,d,e;
2   CFunction pow;
3
4   Local test = pow(a+b+c+d,30);
5   Argument pow;
6   id a = -b+5;
7   id c = -d+e;
8   EndArgument;
9
10  id pow(?k,a?) = exp_(?k,a);
11  Print;
12  .end
```

```
Time =      0.00 sec    Generated terms =         31
          test          Terms in output =         31
                        Bytes used      =       1068
```

Compare the timing/memory usage with the previous approach.

Built-in Functions: dd_

- dd_ is a totally symmetric tensor represented as a sum of symmetrized product of Kronecker deltas (d_):

```
1  #-
2  AutoDeclare Vector v;
3  Local F = dd_(v1,v2,v3,v4);
4  Print;
5  .end
```

Output:

```
F =
    v1.v2*v3.v4 + v1.v3*v2.v4 + v1.v4*v2.v3;
```

Fun quest:

$$\langle\Omega|T\{\phi(x_1)...\phi(x_n)\}|\Omega\rangle = \frac{\langle 0|T\left\{\phi_0(x_1)...\phi_0(x_n)e^{i\int d^4x\mathcal{L}_i[\phi_0]}\right\}|0\rangle}{\langle 0|T\left\{e^{i\int d^4x\mathcal{L}_i[\phi_0]}\right\}|0\rangle}$$

Demonstrate that in the S-matrix vacuum bubble diagrams cancel at any order of a perturbation theory.

Built-in Functions: e_

- e_ is a Levi-Civita Tensor

```
1   #-
2   Dimension 3;
3   Indices i,j,k,p,q,r;
4   Local test = e_(i,j,k) * e_(p,q,r);
5   *Wrong
6   *id r = k;
7
8   contract;
9   Print;
10  .end
```

How can we make a replacement $r \to k$ to obtain the following output?

```
test =
      d_(i,p)*d_(j,q) - d_(i,q)*d_(j,p);
```

What would you do if we don't know how many indices are inside e_?

Built-in Functions: g_

- g_ objects represent the Dirac gamma matrices

```
1  #-
2  Dimension D;
3  Index i1,i2;
4  Local test = g_(1, i1)*g_(1,i2);
5  tracen 1;
6  Print;
7  .end
```

Produces:

```
test =
   4*d_(i1,i2);

0.00 sec out of 0.00 sec
```

Show that:

$$\mathrm{tr}\left(\gamma^{\mu}\gamma^{\nu}\gamma^{\rho}\gamma^{\sigma}\gamma^{5}\right) = -4i\epsilon^{\mu\nu\rho\sigma}.$$

$$
\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
1 & & 5 & & 10 & & 10 & & 5 & & 1
\end{array}
$$

- Build the Pascal's triangle from top to bottom
- Define a procedure which can extract an element at the depth i with the column number j
- Calculate the sum of all elements at the depth i.
- i and j are integers between 100 and 1000.

Thanks for your attention!

Karlsruhe Institute of Technology