# reFORM: designing a new symbolic manipulation toolkit

**Ben Ruijl**

ETH Zurich, Wolfgang-Pauli-Str. 27, 8093 Zürich, Switzerland

E-mail: bruijl@phys.ethz.ch

**Abstract.** Since modern-day multi-loop Feynman diagram computations often require manipulating billions of terms, taking up terabytes of memory, a powerful symbolic manipulation toolkit (SMT) is essential. The de facto solution is FORM, but it has several shortcomings. In this work we present reFORM, a new SMT in early development, that will handle the same workload as FORM but does not have its shortcomings. We showcase some features of reFORM, including Python and C APIs. Finally, we provide benchmarks for polynomial GCD computations, which show that reFORM often outperforms its competitors. A link to the source code of the technical preview version is provided.

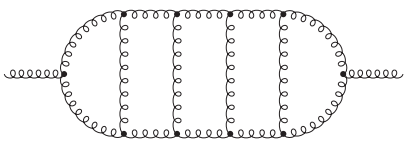
## 1. Introduction

With the rise of computers in the 1960s, computer algebra immediately became an essential tool for theoretical high energy physics. In 1963 the computer program SCHOONSCHIP was created by Veltman for the symbolic computation of the early Standard Model [1]. The program had to process an expression with 50 000 terms, which at the time could only be done by storing intermediate results on tape.

The development of the symbolic manipulation toolkit FORM [2] was started in 1984. It was designed to efficiently handle large expressions.

The fields of computer algebra and particle physics have developed hand in hand. Even though new computational methods have been designed and the hardware has had spectacular improvements, combinatorial and algorithmic challenges have not disappeared due to the need for more precision. In the 1960s a calculation involving proton interactions was considered a success if the order of magnitude agreed with the experiment. Nowadays, the goal is to achieve 1% accuracy. As a result, we will encounter expressions with billions of terms, taking up more than a terabyte of memory.

Examples of these modern huge calculations are computations of the five-loop beta function [3, 4] and four-loop and five-loop splitting functions [5, 6] Just a single fully gluonic five-loop diagram



represents 12 029 521 scalar integrals and needs a terabyte of disk space to compute.

The only symbolic manipulation toolkit to our knowledge that can handle such a workload is a modern-day version of FORM [7, 8].

In this work we present a first technical preview of reFORM [9], a new symbolic manipulation toolkit that aims to be as fast and feature-rich as FORM while alleviating some of FORM's shortcomings.

The outline of this paper is as follows. In section 2 we present related work. In section 3 we present reFORM and show some of its language features. In section 4 we present benchmarks of the multivariate polynomial GCD routines. Finally, we provide our conclusion in section 5.

## 2. Related work

FORM is the only symbolic manipulation toolkit that we are aware of that can handle expressions with billions of terms. The key feature that allows this is that terms are processed one at a time. When a term has been processed, it is written to disk so that the computer does not run out of memory. When all terms are processed, they are sorted using an n-way merge sort. Using this setup, expressions that do not fit in memory can be processed efficiently. FORM uses a (compressed) linear memory model to describe terms, which is good for caching and has a low-memory footprint. It is also equipped with a powerful pattern matcher that allows for terms to be transformed, and it has features specifically designed for physicists, such as gamma matrices, non-commutative functions, tensors and indices. An example FORM program is displayed in listing 1.

```
S x,y,z,a,n,n1;
L F = (1+x+y+z)^50;
id a?^n?*y^n1? = y^(n+n1); * executed term by term
.sort; * terms are merged and sorted
```

**Listing 1.** An example FORM program

Since FORM has been developed incrementally over more than two decades, its design is not always equipped to handle modern-day requirements conveniently. One of the limitations of FORM is that there is a predefined limit to the memory size of a term. This limit cannot be set to be the size of the memory itself, since several buffers scale with this number, resulting in small expressions being padded to the maximum. FORM is also written in optimized C (and some C++), which makes it prone to memory bugs. Indeed, every year memory bugs are fixed that take months of debugging to pinpoint. Even for the latest version, there is a chance that the program crashes after many weeks of running due to memory corruption. Due to its internal design, the pattern matcher has some limitations as well, for example when using nested ranged wildcards (`?a`, etc.). Futhermore, there is a lack of documentation and there are some seemingly valid design patterns that do not work or result in a crash, but it is unclear why. Experienced FORM users have internalized workarounds that one "needs to know", but this unexpected behaviour provides a challenge for new users. Most of these issues will never be fixed, since it requires rewriting complicated existing code.

Another aspect that makes FORM challenging is that the control flow is difficult to follow. The majority of the logic for algorithms is written in the preprocessor. The preprocessor consists of text-based substitutions that occur before the compiler compiles the code. Furthermore, most (but not all!) operations in a module are executed term-by-term in an implicit loop. There is no distinction between a global scope and a term-by-term scope. Finally, when a statement such as an `identify` statement generates more than one term, each output term is processed depth-first. These three features lead to complicated control flow. An example of a FORM program with complicated flow is given in listing 2.

```
#do i=1,5
    .sort
    #do j=1,‘i’
        L F‘j’‘i’ = x‘j’+x^2;
        #write "test2"
    #enddo
    id x = 1 + x;
    Print "%t";
    #write "test3"
#enddo
```

**Listing 2.** FORM program with complex control flow.

## 3. reFORM

reFORM is a new symbolic manipulation toolkit that aims to handle expressions with billions of terms. Its main design philosophy of handling terms one by one is inherited from FORM. It has five major goals: it should (1) be competitive to FORM in terms of performance and memory usage, (2) be built in a language that prevents memory bugs, (3) have a transparent scripting language (front-end) for the user, (4) have proper documentation, and (5) have a C and Python API so that it is usable as a library.

### *3.1. Rust*

Programs involving higher-loop computations generally run for weeks on a cluster, which makes memory bugs and random crashes tedious for the user and hard to debug for the programmer. To prevent memory bugs and concurrency issues, we chose to write reFORM in Rust. Rust is a relatively young systems programming language supported by Mozilla that guarantees memory safety (including race conditions, etc.) and zero-cost abstractions at compile time. At runtime there is no overhead. Below we show an example of a simple C++ program with a hard to spot memory bug (for beginners) and its Rust equivalent:

```
#include <vector>
int main()
{
  std::vector<int> a = {1,2,3};
  int* ref = &a[0];
  a.push_back(4);
  *ref = 5;
}
```

**Listing 3.** A C++ program with a memory bug

```
fn main() {
    let mut a = vec![1,2,3];
    let b = &mut a[0];
    a.push(4);
    *b = 5;
}
```

**Listing 4.** A similar Rust program will not compile

The C++ program will run and will almost never crash (since the vector is likely not moved), but it could. The Rust compiler will throw the error: `cannot borrow ‘a‘ as mutable more than once`.

Another feature that is used throughout reFORM is Rust's pattern matching:

```
enum Number {
    SmallInt(isize),
    ...
}
enum Expression {
```

```
    Number(Number),
    ...
}
...

if x == Expression::Number(Number::SmallInt(5)) {
    ...
}
```

**Listing 5.** Rust pattern matcher

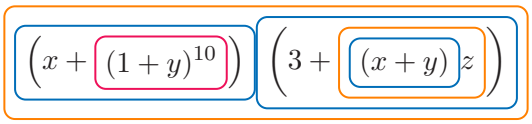
In the snippet above the variable 'x', which is of type expression, is compared to an expression that is the number 5. Pattern matching allows for concise coding of algebraic operations.

### *3.2. Language features*

reForm is inspired by Form for its main design: each term is processed one after another in what is called a *module*. At the end of a module, all resulting terms are sorted (potentially on disk). In Form these modules are designated by a `.sort` instruction at the end of the module and there is no distinction between a global scope, where expressions are defined, and a local scope that only applies to terms (the implicit term loop). In reForm the two scopes will be clearly separated.

Many per-term operations, such as identify statements and expansion have the potential to create lots of terms. It is not guaranteed that the result of these transformations fit in memory. Therefore, instead of generating the entire result of an operation at once, each operation is an iterator that yields one term of the output at a time. This term is then processed by the rest of the module.

Below is an example of an expansion operation, and its deconstruction into iterators of three different types:




where products of factors (orange) are handled by a Cartesian product iterator, subexpressions (blue) are handled by a sequence iterator, and an expression to the power of a positive integer (red) is handled by a binomial iterator.

At the moment the memory model is tree-like, but this may change in the future, as it increases the memory usage of a term and there are some hints that it is the cause of cache misses.

### *3.3. reFORM examples*

In reForm the default scope is the global scope. The per-term part of the program is captured in an `apply` block.

```
expr F = f(2) + f(3);
apply {
    id f(x?) = f(x? + 1);
}
print F;
```

**Listing 6.** A reForm example of an apply-block

Listing 6 shows the creation of expression $F = f(2) + f(3)$, and the transformation $f(x) \to f(x+1)$ that is applied to every term (the `?` needs to be used on the right hand side as well, contrary to Form). Finally, the entire expression is printed.

In the listing 7 we use variables (objects starting with a $) and a for-loop outside of a module. No preprocessor is needed as was required in FORM. Additionally, the for-loop can perform mathematical manipulations on its bounds, since it's considered a mathematical expression. The for-loop outside of a module will be unrolled by the compiler.

```
expr F = f(2+y,x*y);
$v = 10;
for $i in 1..($v * 2) {
    apply {
        id f($i+x?,x?*y?) = f(x?);
    }
}
print;
```

**Listing 7.** A reFORM example

In listing 8 we show a more complex pattern matching that is only partly supported by FORM:

```
expr F = f(1,2,f(x1*x2,x3*x4,x5*x6),x1*x3,x3*x5);
apply {
    id all f(1,2,f(?a,x1?*x2?,?b),?c,x1?*x3?) =
            f(x1?,x2?,x3?);
}
```

**Listing 8.** Complex pattern matching in reFORM

Listing 8 yields `f(x3,x4,x5)+f(x5,x6,x3)`.

Variables in reFORM act similarly to functions, since they can be multi-indexed by any expression. This allows for dynamic storage, as displayed in listing 9.

```
for $i in 1..3 {
    $a[$i+x,2] = $i;
}

$b = $a[2+x,4] + f(x);

inside $b {
    id f(x?) = $a[1+x?,2];
    id $a[x?,?a,y?] = $a[x?,?a,y?-2];
}

print $b;
```

**Listing 9.** Variable indexing in reFORM

### *3.4. API*

Finally, reFORM can be used as a module in other programming languages. At the moment we provide a C and Python API.

In listing 10 we demonstrate how the Python API can be used to manipulate expressions. It also shows the polynomial class, which uses highly optimized rational arithmetic.

```
import reform

vi = reform.VarInfo()
a = reform.Expression("x+y^2", vi)
```

```
b = reform.Expression("z + y", vi)
c = a * b

print("c: ", c, ", c expanded: ", c.expand())

d = c.expand().id("x", "1+w", vi)
print("Substituted x->1+w: ", d)

# Polynomial API
a = reform.Polynomial("1+x*y+5", vi)
b = reform.Polynomial("x^2+2*x*y+y", vi)
g = a + b
ag = a * g
bg = b * g
print(gcd(ag, bg))

# Rational polynomial API
rat = reform.RationalPolynomial(ag, bg)
print(rat)
```

**Listing 10.** reFORM Python API

## 4. Multivariate polynomial GCDs

One important aspect of higher-loop computations is rational arithmetic. Especially in integration-by-parts algorithms, the coefficients of the intermediate results can easily grow to several hundred megabytes in size. The main bottleneck of arithmetics with multivariate rational polynomials is the computation of greatest common divisors (GCDs). In reFORM, we provide first-class support for GCD computations with an efficient implementation of the LINZIP[10] algorithm, with several optimizations such as improved GCD bounds estimation and optimization of the ordering of the variables in the monomial. It also supports rewriting the polynomial with Horner schemes and common subexpression elimination to handle polynomials with millions of terms more efficiently [11]. Further details will be provided in a future publication. In table 1 we compare the performance of reFORM with four commonly used GCD implementations: Form 4.2 [7], Rings 2.5.2 [12], Mathematica 11.3[1], and Fermat 6.21.

We compute the GCD of the following dense polynomials:

$$\begin{aligned} a &= (1 + 3x_1 + 5x_2 + 7x_3 + 9x_4 + 11x_5 + 13x_6 + 15x_7)^7 - 1 \\ b &= (1 - 3x_1 - 5x_2 - 7x_3 + 9x_4 - 11x_5 - 13x_6 + 15x_7)^7 + 1 \\ g &= (1 + 3x_1 + 5x_2 + 7x_3 + 9x_4 + 11x_5 + 13x_6 - 15x_7)^7 + 3 \\ ag &= ag, bg = bg, D = \mathrm{GCD}(ag, bg) = g, D_{\mathrm{simple}} = \mathrm{GCD}(ag + 1, bg) = 1 \end{aligned} \tag{1}$$

and of randomly generated polynomials in regimes that are common in physics calculations. They are classified as $R(v, d, t, p)$, where $v$ is the number of variables, $d$ the degree, $t$ the number of terms, and $p$ the maximum coefficient power. Each result per regime has been averaged over 200 random polynomials.

Table 1 shows that reForm is often among the fastest (or the fastest) for these benchmarks. Its performance also doesn't degrade by orders of magnitude in some of the tested regimes, as happens with Mathematica and Fermat.

[1] There appears to be a regression in Mathematica 11.3, as the $D$-polynomial did not yield a result in 9000 seconds.

| | reForm | Form | Rings | Mathematica | Fermat |
|---|---|---|---|---|---|
| $D$ | 58.27 | 82.87 | 86.77 | $> 9000$ | 1241.82 |
| $D_{\text{simple}}$ | 0.31 | 0.20 | 0.13 | 1.39 | 0.03 |
| $R(2, 50, 1000, 50)$ | 0.32 | 0.88 | 0.91 | 0.25 | 0.19 |
| $R(5, 30, 50, 50)$ | 0.17 | 0.80 | 0.20 | 1.98 | 10.60 |
| $R(5, 30, 100, 50)$ | 0.54 | 0.81 | 1.02 | 11.50 | 22.65 |
| $R(10, 10, 100, 50)$ | 0.83 | 0.85 | 3.12 | 9.62 | 37.12 |

**Table 1.** Computed on a Ryzen 2700X with 2x8 GB of 3000Mhz memory. The timings are in seconds and are averaged over 200 random polynomials for each regime $R$.

## 5. Conclusion

reForm is a new symbolic manipulation toolkit that aims to handle expressions with billions of terms that take up terabytes of disk space. It processes expressions term-by-term to prevent running out of memory. Its goals are to be easier to use, more stable and as fast as Form. Additionally, we expose C and Python APIs. Even though reForm is in early development, the technical preview release offers full support for multivariate polynomial GCDs, and shows that it often performs better than existing computer algebra toolkits.

The reForm source code [9] can be obtained on Github under the MIT license: `http://github.com/benruijl/reform`.


## Acknowledgements

I would like to thank Jos Vermaseren, Takahiro Ueda, Joshua Davies, and Stanislav Poslavsky for valuable discussions and insights. Furthermore, I would like to thank Takahiro Ueda for providing a polynomial GCD testing suite.

This project has received funding from the European Research Council (ERC) under grant agreement No 694712 (PertQCD)


## References


[1] Veltman M J G and Williams D N 1991 (*Preprint* `hep-ph/9306228`)
[2] Vermaseren J A M 2000 (*Preprint* `math-ph/0010025`)
[3] Herzog F, Ruijl B, Ueda T, Vermaseren J A M and Vogt A 2017 *JHEP* **02** 090 (*Preprint* `1701.01404`)
[4] Baikov P A, Chetyrkin K G and Kühn J H 2017 *Phys. Rev. Lett.* **118** 082002 (*Preprint* `1606.08659`)
[5] Ruijl B, Ueda T, Vermaseren J A M and Vogt A 2017 *JHEP* **2017** 1–50 ISSN 1029-8479 URL `http://dx.doi.org/10.1007/JHEP06(2017)040`
[6] Herzog F, Moch S, Ruijl B, Ueda T, Vermaseren J A M and Vogt A 2019 *Phys. Lett.* **B790** 436–443 (*Preprint* `1812.11818`)
[7] Ruijl B, Ueda T and Vermaseren J 2017 (*Preprint* `1707.06453`)
[8] Kuipers J, Ueda T, Vermaseren J A M and Vollinga J 2013 *Comput. Phys. Commun.* **184** 1453–1467 (*Preprint* `1203.6543`)
[9] Ruijl B 2019 benruijl/reform: First technical preview URL `https://doi.org/10.5281/zenodo.3229823`
[10] de Kleine J, Monagan M and Wittkopf A 2005 *Proceedings of the 2005 International Symposium on Symbolic and Algebraic Computation* ISSAC '05 (New York, NY, USA: ACM) pp 124–131 ISBN 1-59593-095-7 URL `http://doi.acm.org/10.1145/1073884.1073903`
[11] Ruijl B, Vermaseren J, Plaat A and van den Herik H J 2014 Why Local Search Excels in Expression Simplification (*Preprint* `1409.5223`) URL `http://arxiv.org/abs/1409.5223`
[12] Poslavsky S 2019 *Comput. Phys. Commun.* **235** 400–413 (*Preprint* `1712.02329`)