# reFORM: designing a new symbolic manipulation toolkit

Ben Ruijl

ETH Zurich

Mar 14, 2019

# Computer algebra and HEP

Symbolic manipulation is an important part of HEP:

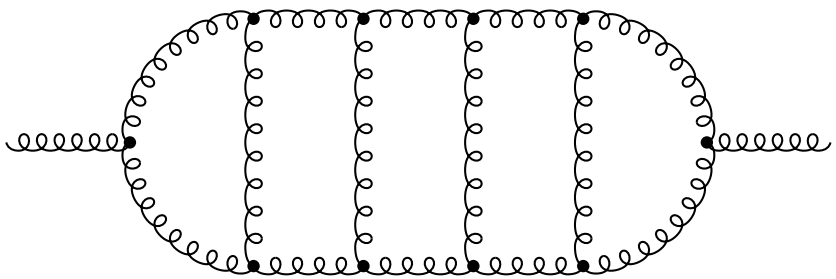
- Schoonschip [Veltman '63]
- FORM [Vermaseren '89]
- FORM 4.2.1 [Ueda, Vermaseren, BR '17] [Takahiro's talk]

Computations with billions of terms:

- Five-loop beta function [Chetyrkin, Baikov, Kühn, Herzog, Vermaseren, Vogt, Ueda, BR, Schröder, Maier, Luthe, Marquard,...]
- Four-loop and five-loop splitting functions [Vermaseren, Vogt, Ueda, BR]

# Computational blow-up

- Troublesome five loop diagram:



- Represents 12 029 521 scalar integrals!
- Computation needs a terabyte of disk space
- Requires solid symbolic manipulation toolkit

# FORM: the good

- Memory is not an issue: terms will be streamed to/from disk
- Very fast, low memory usage
- Powerful pattern matching
- Features for physicists (gamma matrices, indices, . . . )
- Open source with issue tracker

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

# FORM: the bad

- Limited term length
- Memory bugs (written in C)
- Strange limits on pattern matching
- Sub-par user documentation
- Often workarounds required that one “needs to know”

Some of these issues will never be fixed!

# FORM: the ugly

- Preprocessor is used for 90% of the logic
- Flow of the program is confusing: preprocessor, implicit term loop, iterators

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

# reFORM goals

Goals:

- Design a program that takes the good parts of FORM
- Use a modern language that prevents memory bugs
- Modernize the FORM language, make it more transparent
- Fix shortcomings that are hard to fix in FORM itself
- Introduce APIs for multiple languages (Python, C, ...)
- Improve handling of polynomials (GCD, arithmetic)

# Problems with C/C++

- Writing correct C++ is absurdly difficult
- I am not talented enough to write a safe C++ program

```
1 #include <vector>
2 int main()
3 {
4   std::vector<int> a = {1,2,3};
5   int* ref = &a[0];
6   a.push_back(4);
7   *ref = 5;
8 }
```

# Rust

Advantages:

- **Compile-time** guaranteed memory safety
- Race conditions are impossible
- No undefined behaviour
- Pattern matching
- Built-in package manager
- Supported by Mozilla

# Borrow checker I

```
1 fn main() {
2     let mut a = vec![1,2,3];
3     let b = &mut a[0];
4     a.push(4);
5     *b = 5;
6 }
```

# Borrow checker I

```
1 fn main() {
2     let mut a = vec![1,2,3];
3     let b = &mut a[0];
4     a.push(4);
5     *b = 5;
6 }
```

```
error[E0499]: cannot borrow ‘a‘ as mutable more than once
    at a time --> src/main.rs:4:3
  |
3 |   let b = &mut a[0];
  |                - first mutable borrow occurs here
4 |   a.push(4);
  |   ^ second mutable borrow occurs here
5 |   *b = 5;
  |   ------ first borrow later used here
```

# Borrow checker II

```
1 fn main() {
2     let mut a = vec![1,2,3];
3     let b = &a[0];
4     a.push(4);
5     println!("{}", b);
6 }
```

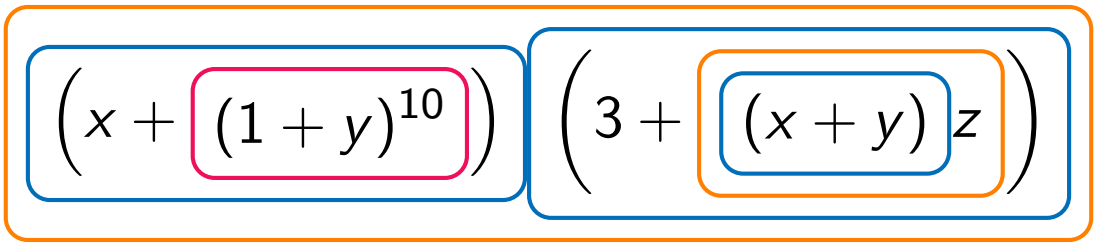
```
error[E0502]: cannot borrow ‘a‘ as mutable because it is
    also borrowed as immutable --> src/main.rs:4:3
  |
3 |   let b = &a[0];
  |           - immutable borrow occurs here
4 |   a.push(4);
  |   ^^^^^^^^^ mutable borrow occurs here
5 |   println!("{}", b);
  |                   - immutable borrow later used here
```

# Pattern matching

```
1  enum Number {
2      SmallInt(isize),
3      ...
4  }
5  enum Expression {
6      Number(Number),
7      ...
8  }
9  ...
10
11 if x == Expression::Number(Number::SmallInt(5)) {
12     ...
13 }
```

# Internals

- Almost every operation is an iterator, since the result may not fit in memory
- Expansion operation:




- Product of factors: Cartesian product iterator
- Subexpressions: sequence iterator
- Powers of positive integer: binomial iterator

# reFORM example

- Clear distinction between global scope and term-by-term evaluation by `apply` block
- No preprocessor needed for logic

```
1 expr F = f(2+y,x*y);
2 $v = 10;
3 for $i in 1..($v * 2) {
4     apply {
5         id f($i+x?,x?*y?) = f(x?);
6     }
7 }
8 print;
```

# Extended pattern matcher

```
1 expr F = f(1,2,f(x1*x2,x3*x4,x5*x6),x1*x3,x3*x5);
2 apply {
3     id all f(1,2,f(?a,x1?*x2?,?b),?c,x1?*x3?) =
4             f(x1?,x2?,x3?);
5 }
```

yields

```
1 f(x3,x4,x5)+f(x5,x6,x3)
```

# Indexing variables

- Variables behave like tables/functions
- Indexible with any expression

```
1  for $i in 1..3 {
2      $a[$i+x,2] = $i;
3  }
4
5  $b = $a[2+x,4] + f(x);
6
7  inside $b {
8      id f(x?) = $a[1+x?,2];
9      id $a[x?,?a,y?] = $a[x?,?a,y?-2];
10 }
11
12 print $b;
```

# Python API

```python
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)
```

# Python API for polynomials

```
1  import reform
2
3  vi = reform.VarInfo()
4
5  a = reform.Polynomial("1+x*y+5", vi)
6  b = reform.Polynomial("x^2+2*x*y+y", vi)
7  g = a + b
8
9  ag = a * g
10 bg = b * g
11 print(gcd(ag, bg))
12
13 rat = reform.RationalPolynomial(ag, bg)
14 print(rat)
```

# Polynomial GCDs

- First class polynomial gcd support
- Can easily be used as a library: no overhead from string conversions!
- Often much faster than FORM and Fermat [Lewis '85]
- Seems competitive with Rings [Polavsky '18]

# Conclusions

- reFORM is a new symbolic manipulation toolkit
- In early development
- Aims to be be easier to use than FORM
- Should be able to process terabytes of terms
- API for Python and C
- Polynomial GCD library already working

Source code: `http://github.com/benruijl/reform`