Symbolica

The future of FORM

Ben Ruijl

Apr 13, 2023

Ruijl Research

What if we started from scratch?

The good parts of FORM

- Memory is not a bottleneck
  - Terms are processed one by one (depth first)
  - Terms can be streamed to/from disk
- Very fast, low memory usage
- Progress updates
- Free to use
- Features for particle physicists (gamma algebra, vectors, etc.)

Huge accomplishment

FORM is an extremely impressive piece of software whose algorithms survived for decades!

The bad I

- Counterintuitive control flow
- Text-based preprocessor is used for logic
- Hard to act at the expression level due to implicit loop over terms

```
#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
```

The bad II

- No namespaces
- Term length limitations (`MaxTermSize` and company)
- Lack of data structures, hashmaps etc have to be emulated
- Often workarounds required that one “needs to know”
- No native factorisation: $(x + 1)(x + 2)$ will be expanded
- Bugs that will never be fixed
- IO to other languages is not great
- Not possible to use as a library

The bad III

- Code written when compilers and system allocators could not be trusted
- Mixing of logic and expression representation
- Pointer and offset hacks that are no longer needed?

The Mathematica bad

- Closed ecosystem and expensive
- Limits the number of cores!
- Functions are black boxes
- No progress updates
- Poor IO to other languages
- Poor scaling to large problems

Symbolica

- Symbolica is a new computer algebra system
- Focus on flexibility and ease of use in existing projects
- Should be easy to pass data to and from FORM, Mathematica, etc.
- Open development on Zulip and Github
- Blog posts and documentation on `https://symbolica.io`
- Community supported

Which language?

> *While many experienced programmers can write correct systems-level code, it's clear that no matter the amount of mitigations put in place, it is near impossible to write memory-safe code using traditional systems-level programming languages at scale.*

- Microsoft Security Research Center

```
1  int* test(int* old, bool use_new) {
2    std::vector<int> a = {1, 2, 3};
3    int* b = &a[0];
4    a.push_back(4);
5    *b = 5; // BUG 1
6    if use_new { return b; } else { return old; } // BUG 2
7  }
```

Which language?

> *While many experienced programmers can write correct systems-level code, it's clear that no matter the amount of mitigations put in place, it is near impossible to write memory-safe code using traditional systems-level programming languages at scale.*

- Microsoft Security Research Center

```
1  int* test(int* old, bool use_new) {
2    std::vector<int> a = {1, 2, 3};
3    int* b = &a[0];
4    a.push_back(4);
5    *b = 5; // BUG 1
6    if use_new { return b; } else { return old; } // BUG 2
7  }
```

Rust saves the day

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

Gives compilation error:

```
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
```

Also work for multi-threaded code!

Rust saves the day

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

Gives compilation error:

```
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
```

Also work for multi-threaded code!

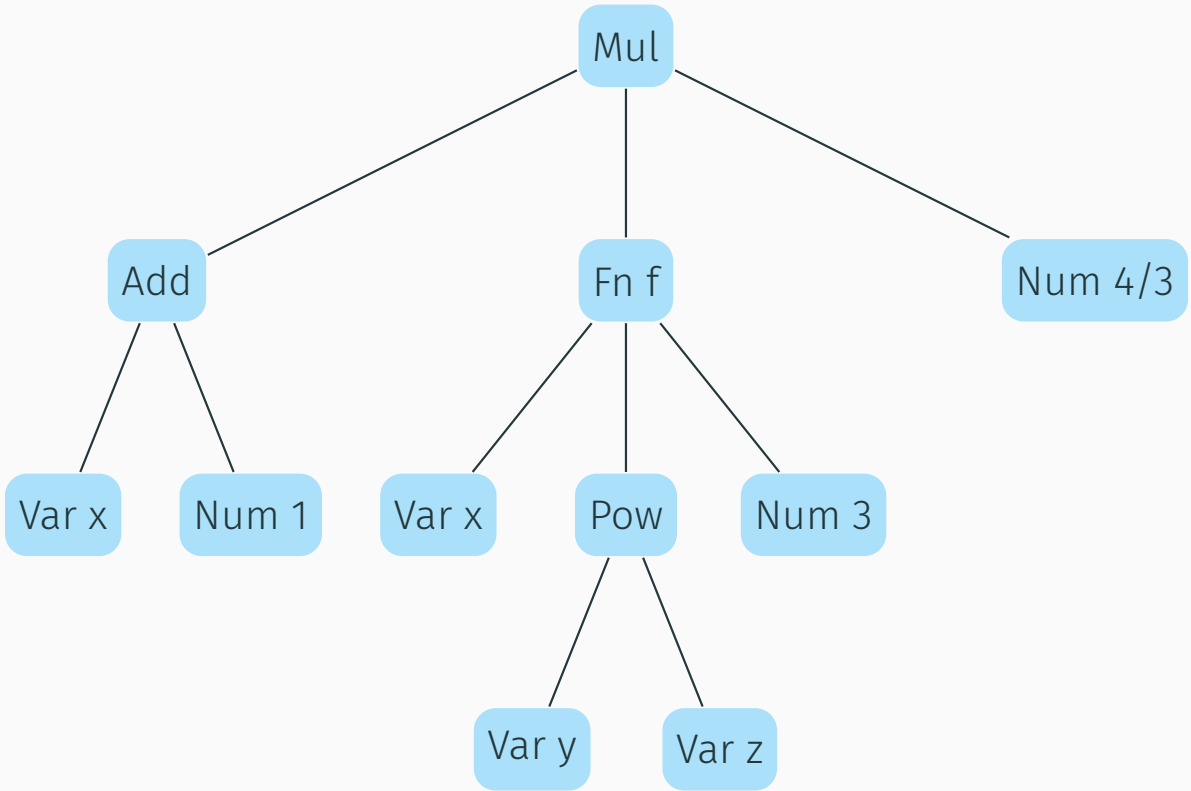
Expressions

Expressions in FORM:

- `symbol^int`, functions, subexpression, dot products, vectors, indices
- A function to an integer power is written out completely
- Any other power is converted: `x^x -> (x)^(x)` (36 to 48 bytes)
- `x^n?` does not match `x^x` but `x^(n?)` does
- Subexpressions are automatically expanded

Expressions in Symbolica

- Atoms: Mul, Add, Pow, Var, Fun, Num
- $1/x$ is represented as x^{-1} and $-x$ is represented as $-1 * x$
- Tensor support could be done through atom "Indexed"
- Representation of `(x+1)*f(x,y^z,3)*4/3`:



Expression representation

The FORM C code mixes logic and the memory representation:

```
1  for ( i = 1; i <= count; i++ ) {
2    m = start;
3    while ( m < stop ) {
4      r = m + m[1];
5      j = *r++;
6      if ( j > 1 ) {
7        while ( --j > 0 ) {
8          if ( *r == i ) goto RightNum;
9          r++;
10       }
11       m = r;
12       continue;
13     }
```

- A better way: create zero-cost abstractions
- In Rust *traits* describe what functions a `struct` should have

Expression representation II

An `AtomView` is an enum of borrowed data:

```
pub enum AtomView<'a, P: Atom> {
    Num(P::N<'a>),
    Var(P::V<'a>),
    Fun(P::F<'a>),
    Pow(P::P<'a>),
    Mul(P::M<'a>),
    Add(P::A<'a>),
}
```

`P::F` is any `struct` that satisfies the following constraints:

```
pub trait Fun<'a> {
    type P: Atom;
    type I: Iterator<Item = AtomView<'a, Self::P>>;

    fn get_name(&self) -> Identifier;
    fn get_nargs(&self) -> usize;
    fn cmp(&self, other: &Self) -> Ordering;
    fn into_iter(&self) -> Self::I;
}
```

Expression representation II

An `AtomView` is an enum of borrowed data:

```
pub enum AtomView<'a, P: Atom> {
    Num(P::N<'a>),
    Var(P::V<'a>),
    Fun(P::F<'a>),
    Pow(P::P<'a>),
    Mul(P::M<'a>),
    Add(P::A<'a>),
}
```

`P::F` is any `struct` that satisfies the following constraints:

```
pub trait Fun<'a> {
    type P: Atom;
    type I: Iterator<Item = AtomView<'a, Self::P>>;

    fn get_name(&self) -> Identifier;
    fn get_nargs(&self) -> usize;
    fn cmp(&self, other: &Self) -> Ordering;
    fn into_iter(&self) -> Self::I;
}
```

Tree walk agnostic of the representation

```
fn tree_crawl<'a, P: Atom>(atom: AtomView<'a, P>) {
    match atom {
        AtomView::Num(_) | AtomView::Var(_) => println!("{:?}", atom),
        AtomView::Fun(f) => {
            println!("Fun {:?}", f.get_name());
            for a in f.into_iter() {
                tree_crawl(a);
            }
        }
        AtomView::Pow(p) => {
            let (base, exp) = p.get_base_exp();
            println!("Pow");
            tree_crawl(base);
            tree_crawl(exp);
        }
        AtomView::Mul(m) => {
            println!("Mul");
            for a in m.into_iter() {
                tree_crawl(a);
            }
        }
        ...
```

Compact linear representation I

- A compressed linear format: `tag, size, data`
- Tag 1 is a number, tag 2 a variable, etc
- Packing of two numbers: `bit flag, num1, num2`

Meaning	Bit flag
U8 NUM	0000 0001
U16 NUM	0000 0010
U32 NUM	0000 0011
U64 NUM	0000 0100
FIN NUM	0000 0101
ARB NUM	0000 0111

Meaning	Bit flag
U8 DEN	0001 0000
U16 DEN	0010 0000
U32 DEN	0011 0000
U64 DEN	0100 0000
ARB DEN	0111 0000
SIGN	1000 0000

Still unused bits available to code a rational polynomial

Compact linear representation I

- A compressed linear format: `tag, size, data`
- Tag 1 is a number, tag 2 a variable, etc
- Packing of two numbers: `bit flag, num1, num2`

Meaning	Bit flag
`U8 NUM`	0000 0001
`U16 NUM`	0000 0010
`U32 NUM`	0000 0011
`U64 NUM`	0000 0100
`FIN NUM`	0000 0101
`ARB NUM`	0000 0111

Meaning	Bit flag
`U8 DEN`	0001 0000
`U16 DEN`	0010 0000
`U32 DEN`	0011 0000
`U64 DEN`	0100 0000
`ARB DEN`	0111 0000
`SIGN`	1000 0000

Still unused bits available to code a rational polynomial

Compact linear representation II

- Compression used throughout: all variable names are packed
- The first 256 variable names only take up 2 bytes
- Function name and number of args are packed together, often taking up 3 bytes
- For example `f(x,2/5)` is coded in 15 bytes:

[3, 10, 0, 0, 0, 17, 1, 2, 2, 1, 0, 1, 17, 2, 5]

Byte	Meaning	Byte	Meaning
3	Function tag	1	length 1 num and 0 den
10, 0, 0, 0	Length	0	Name ‘x‘
17	Len 1 num and len 1 den	1	Number tag
1	Name ‘f‘	17	Len 1 num and len 1 den
2	Number of arguments	2	Numerator
2	Variable tag	5	Denominator

2x shorter expressions than FORM and 8x shorter than Mathematica

Compact linear representation II

- Compression used throughout: all variable names are packed
- The first 256 variable names only take up 2 bytes
- Function name and number of args are packed together, often taking up 3 bytes
- For example `f(x,2/5)` is coded in 15 bytes:

[3, 10, 0, 0, 0, 17, 1, 2, 2, 1, 0, 1, 17, 2, 5]

Byte	Meaning	Byte	Meaning
3	Function tag	1	length 1 num and 0 den
10, 0, 0, 0	Length	0	Name ‘x‘
17	Len 1 num and len 1 den	1	Number tag
1	Name ‘f‘	17	Len 1 num and len 1 den
2	Number of arguments	2	Numerator
2	Variable tag	5	Denominator

2x shorter expressions than FORM and 8x shorter than Mathematica

Workspace

- In FORM the workspace is globally available and is mutable
- This can lead to dangerous bugs
- In Symbolica, the workspace has a list of owned Atoms (a vector of vector of bytes)

```
let mut handle: Handle<OwnedAtom> = workspace.new_atom();
let new_atom: &mut OwnedAtom = handle.get_buf_mut();
new_atom.from_view(&atom);
```

- The memory is automatically returned to the workspace when `handle` goes out of scope

Coefficients

- A number in Symbolica is either an integer quotient, a finite field entry or a rational polynomial
- The default coefficient is an integer quotient
- Expand the coefficient ring by x:

```
x*y+x^2*y * [2] -> y * [2 x + 2 x^2]
```

- Normalisation moves x into the coefficient, should it also expand $(x+2)^2$?

Bracketing

- Split off expressions based on pattern?
- Bracketing in `f(x?)` on

$$f(x) + f(y) + f(x)x^2 + f(y)y^2$$

  gives `[(f(x), 1 + x^2), (f(y), 1+y^2)]`
- Each expression may be on disk

Control flow

- Symbolica should be a library useful within existing projects
- This means that the functions such as `id` should be standalone
- In FORM, no statement is standalone but is in a recursive chain

```
input = 'f(1,2,3)'
statements = ['id all f(?a,?b) = f(?a)*f(?b);', 'id all f(?a) = 1;', '.sort']

def id_all_statement(lhs, rhs, target_term, next_instruction):
    for match in get_matches(lhs, target_term):
        for r in get_rhs(match, rhs):
            do_next_instruction(r, next_instruction)
```

- No local state is kept in `id all`
- Every Symbolica function should be an iterator / generator

Generators

```
1  def example_generator():
2      for x in range(100):
3          yield x
4
5  r = example_generator()
6  assert(next(r) == 0)
7  assert(next(r) == 1)
```

Self-kept state:

```
1  class A:
2      def __init__():
3          self.counter = 0
4      def next(self):
5          cur = self.counter
6          self.counter += 1
7          return cur
```

The state machine of the pattern matcher is quite complicated!

Generators

```
def example_generator():
    for x in range(100):
        yield x

r = example_generator()
assert(next(r) == 0)
assert(next(r) == 1)
```

Self-kept state:

```
class A:
    def __init__():
        self.counter = 0
    def next(self):
        cur = self.counter
        self.counter += 1
        return cur
```

The state machine of the pattern matcher is quite complicated!

Tree walk generator i

```rust
pub struct AtomTreeIterator<'a, P: Atom> {
    stack: Vec<(Option<usize>, AtomView<'a, P>)>,
}

impl<'a, P: Atom> AtomTreeIterator<'a, P> {
  pub fn new(target: AtomView<'a, P>) -> AtomTreeIterator<'a, P> {
      AtomTreeIterator {
          stack: vec![(None, target)],
      }
  }

  /// Return the next position and atom in the tree.
  pub fn next(&mut self) -> Option<AtomView<'a, P>> {
    while let Some((ind, atom)) = self.stack.pop() {
      if let Some(ind) = ind {
        let slice = match atom {
            AtomView::Fun(f) => f.to_slice(),
            AtomView::Pow(p) => p.to_slice(),
            AtomView::Mul(m) => m.to_slice(),
```

Tree walk generator ii

```
20                  AtomView::Add(a) => a.to_slice(),
21                  _ => {
22                      continue; // not iterable
23                  }
24              };
25
26              if ind < slice.len() {
27                  let new_atom = slice.get(ind);
28                  self.stack.push((Some(ind + 1), atom));
29                  self.stack.push((None, new_atom));
30              }
31          } else {
32              self.stack.push((Some(0), atom));
33              return Some(atom);
34          }
35      }
36
37      None
38    }
39  }
```

New pattern matching I

- FORM pattern matcher has shortcomings and inconsistencies
- `id p1?.p2?*f(p1?.p2?) = 1;` may not match
- `id f(?a,f(?b,?a,?c),?d) = f(?a,f(?b,?c),?d);` may not match
- Not possible to match subset of factors or summands with `?a`
- `id f(x?)*x? = 1;` does not match to `x*y*f(x*y)` even though it matches `x*y` in the function argument!
- Iterate through all matches without replacement (Mathemetica cannot do this either)
- Should be like `regex` in Python: separate matching and replacement
- Should match at any level

New pattern matching II

- Internally there is only one wildcard type, `x_`, that can match *any* subexpression
- `id x_ = 1` applied to `x*y*z` gives `1`
- `id x_*y_ = f(x_,y_)` applied to `x*y*z` gives all bipartitions
- `id x = 5` applied to `f(x)` gives `f(5)`
- Matching `z*x_*y_*f(z_,x_,w_)` to `x*y*z*w*f(x,y,x*y,z)` gives:
  - `x_ = y, y_ = w, z_ = x, w_ = (x*y, z)`
  - `x_ = y, y_ = x * w, z_ = x, w_ = (x*y, z)`
  - `x_ = x * y, y_ = w, z_ = (x, y), w_ = z`

New pattern matching II

- Internally there is only one wildcard type, `x_`, that can match *any* subexpression
- `id x_ = 1` applied to `x*y*z` gives `1`
- `id x_*y_ = f(x_,y_)` applied to `x*y*z` gives all bipartitions
- `id x = 5` applied to `f(x)` gives `f(5)`
- Matching `z*x_*y_*f(z_,x_,w_)` to `x*y*z*w*f(x,y,x*y,z)` gives:
  - `x_ = y,  y_ = w,  z_ = x,  w_ = (x*y, z)`
  - `x_ = y,  y_ = x * w,  z_ = x,  w_ = (x*y, z)`
  - `x_ = x * y,  y_ = w,  z_ = (x, y),  w_ = z`

New pattern matching III

- Matching and replacement is an iterator
- Repeated calls to `id f(x_) = g(x_)` applied to `f(z)*f(f(x))*f(y)` gives:
  - `g(z)*f(f(x))*f(y)`
  - `f(z)*g(f(x))*f(y)`
  - `f(z)*f(g(x))*f(y)`
  - `f(z)*f(f(x))*g(y)`
- Replace-all function replaces all non-overlapping matches with the first mapping it finds: `g(z)*g(f(x))*g(y)`

Restrictions on wildcards

- Restrictions based on:
  - Type (symbol, number, etc.)
  - Length
  - User-provided boolean function on matched expression
  - User-provided boolean function on matched expressions of two wildcards

Matching pattern `f(x_,y_,z_,w_)` to `f(1,2,3,4,5,6,7)` subject to $0 \leq |\mathsf{x}| \leq 2, 0 \leq |\mathsf{y}| \leq 4, |\mathsf{x}| \geq |\mathsf{y}|, z \in \mathbb{P}$:

- `x_ = 1,       y_ = (),     z_ = 2, w_ = (3, 4, 5, 6, 7)`
- `x_ = 1,       y_ = 2,      z_ = 3, w_ = (4, 5, 6, 7)`
- `x_ = (1, 2), y_ = (),     z_ = 3, w_ = (4, 5, 6, 7)`
- `x_ = (1, 2), y_ = (3, 4), z_ = 5, w_ = (6, 7)`

Restrictions on wildcards

- Restrictions based on:
  - Type (symbol, number, etc.)
  - Length
  - User-provided boolean function on matched expression
  - User-provided boolean function on matched expressions of two wildcards

Matching pattern `f(x_,y_,z_,w_)` to `f(1,2,3,4,5,6,7)` subject to $0 \leq |\mathsf{x}| \leq 2, 0 \leq |\mathsf{y}| \leq 4, |\mathsf{x}| \geq |\mathsf{y}|, z \in \mathbb{P}$:

- `x_ = 1,      y_ = (),     z_ = 2, w_ = (3, 4, 5, 6, 7)`
- `x_ = 1,      y_ = 2,      z_ = 3, w_ = (4, 5, 6, 7)`
- `x_ = (1, 2), y_ = (),     z_ = 3, w_ = (4, 5, 6, 7)`
- `x_ = (1, 2), y_ = (3, 4), z_ = 5, w_ = (6, 7)`

Restriction example in Rust

```
restrictions.insert(
  state.get_or_insert_var("y_"),
  vec![
    PatternRestriction::Length(0, Some(4)),
    PatternRestriction::Cmp(
        state.get_or_insert_var("x_"),
        Box::new(|y, x| {
            let len_x = match x {
                Match::Multiple(_, s) => s.len(),
                _ => 1,
            };
            let len_y = match y {
                Match::Multiple(_, s) => s.len(),
                _ => 1,
            };
            len_x >= len_y
        }),
    ),
  ],
);
```

Python API

```python
from symbolica import Expression, Function

x, y, z = Expression("x"), Expression("y"), Expression("z")
f, g = Function("f"), Function("g")
b = Expression.parse("x^2+2*x*y+y")

# python-style function calls
e1 = f(x, f(x,y))*f(5) * b
e2 = e1.expand()

e3 = e2.id(x, z) # x -> z

# matches f(5) since g.w and x.w are wildcards
e4 = e3.id(g.w(x.w), x.w)
print('e4 =', e4)

for i, t in enumerate(e4):
    print('term { }={ }'.format(i, t))
```

Preprocessor

- When used as a library, the programming language itself is the preprocessor!
- Allows for things not possible in FORM, e.g. store expressions in hashmaps

Example from FORM to Python:

```
L F = f(12) + f(10) + f(8);
#do i = 0, 1
  id f(x?{>1}) = f(x - 1) + f(x - 2);
  if (match(f(x?{>1}))) redefine i "0";
  .sort
#enddo
```

Preprocessor

- When used as a library, the programming language itself is the preprocessor!
- Allows for things not possible in FORM, e.g. store expressions in hashmaps

Example from FORM to Python:

```
from symbolica import Expression

x = Expression.parse('f(12) + f(10) + f(8)')
done = False
while not done:
  y = x.id('f(x_) : x_ > 1', 'f(x_ - 1) + f(x_ - 2))')

  done = True
  for term in y:
    if term.match('f(x_): x_ > 1'):
      done = True
      break

  x = y.sort()
```

Preprocessor

- When used as a library, the programming language itself is the preprocessor!
- Allows for things not possible in FORM, e.g. store expressions in hashmaps

Example from FORM to Python:

```
1  from symbolica import Expression
2
3  x = Expression.parse('f(12) + f(10) + f(8)')
4  while any(term.match('f(x_): x_ > 1') for term in x):
5    x = x.id('f(x_) : x_ > 1', 'f(x_ - 1) + f(x_ - 2))').sort()
```

Computational graph

- With more instructions in a module, the recursive nature will become tedious to write with explicit loops over iterators
- Use a computational graph to build a FORM style module

```
L F = f(12) + f(10) + f(8);
#$some_flag = 1;
repeat;
  if (match(f(5)));
    #if `$some_flag' == 1
      id f(5) = f(1);
    #else
      id f(5) = f(2);
    #endif
  else;
    id f(5) = f(4);
  endif;
  id f(x?{>1}) = f(x - 1) + f(x - 2);
  id f(1) = 1;
endrepeat;
.sort
```

Computational graph

- With more instructions in a module, the recursive nature will become tedious to write with explicit loops over iterators
- Use a computational graph to build a FORM style module

```
from symbolica import Expression
from symbolica.module import Module, repeat, ifstatement, identity, match

x = Expression.parse('f(12) + f(10) + f(8)')
some_flag = True
module = repeat(
  ifstatement(match('f(5)'),
    identity('f(5)', 'f(1)') if some_flag else identity('f(5)', 'f(2)'),
    identity('f(5)', 'f(4)')),
  identity('f(x_)', 'f(x_ - 1) + f(x_ - 2))'),
  identity('f(1)', '1')
)
Module.execute(module, x)
```

Funding

- Goal: community funding through university licenses
- Funding will be used for FORM maintenance as well
- Contributors will be reimbursed
- Continuous funding will make it easier to always have at least two developers working on Symbolica

Join development on:

- `https://symbolica.io`
- `https:/reform.zulipchat.com`
- `https://github.com/benruijl/symbolica`

Thank you for your attention.