Unleashing Algebraic Metaprogramming in Julia with Metatheory.jl

Alessandro Cheli, University of Pisa - Philip Zucker, Draper Labs

(a.cheli6@studenti.unipi.it - philzook58@gmail.com)

Metatheory.jl

Metatheory.jl[1] is a general purpose metaprogramming library for Julia, taking advantage of the language reflection to bridge the gap between symbolic mathematics, abstract interpretation, equational reasoning, optimization, composable compiler transforms and homoiconicity. At its core sits *e-graph rewriting*, a fresh approach to term rewriting achieved through an equality saturation algorithm.

https://github.com/0x0f0f0f/Metatheory.jl

[1] Alessandro Cheli. "Metatheory.jl: Fast and Elegant Algebraic Computation in Julia with Extensible Equality Saturation". In: *Journal of Open Source Software* 6.59 (2021), p. 3078. DOI: 10.21105/joss.03078. URL: https://doi.org/10.21105/joss.03078.

What can we do with Metatheory.jl?

E-Graphs + Julia =

- General purpose equational term rewriting on custom types!
- High-level compiler transforms
- Flexible symbolic mathematics
- Custom code optimizers using equational theories
- Flux.jl model optimizers[2]
- DPLL(T) Theorem Proving[3]
- Symbolic regression algorithms
- Floating point error auto fixers (see here)

[2] Yichen Yang et al. *Equality Saturation for Tensor Graph Superoptimization*. 2021. arXiv: `2101.01332 [cs.AI]`.

[3] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

Equations Are Good

Supporting bidirectional (*equational!*) rules in term rewriting opens up a world of opportunities.

Engineering & Physics

- plus, times, cos, sin, exp. $\cos(x)^2 + \sin(x)^2 = 1$
- integrals, derivatives e.g. $\int x^n dx = \frac{x^{n+\mathbf{1}}}{n}$
- matrix and vector algebra e.g. $(A \otimes B) \cdot (C \otimes D) = (A \cdot C) \otimes (B \cdot D)$
- commutators and operators $a_i a_j^\dagger - a_j^\dagger a_i = \delta_{ij}$

Maths

- Algebraic structures such as Groups $a^2 = b^2 = (ab)^2 = e$
- Category theory $f \cdot id_A = f$

Computer Science

- Stream Fusion `map f . map g = map (f . g)`
- Compiler Optimizations `15 * x = x « 4 - 1`
- Query Optimization

Classical Rewriting

- Mathematica, Maxima, SymPy, Symbolics.jl
- Term rewriting is typically destructive and *forgets* the matched left-hand side.
- Equations are oriented (unnaturally?) in one direction.
- Some rules are obviously good as simplifications. $\cos(x)^2 + \sin(x)^2 = 1$
- Apply rules in arbitrary or controlled order - Local minima and looping
- Confluent and terminating rewrite systems - You're golden[4]
- What about non obviously oriented equations? $(a + b) + c = a + (b + c)$

[4] Nachum Dershowitz. "A taste of rewrite systems". In: *Functional Programming, Concurrency, Simulation and Automated Reasoning*. Springer. 1993, pp. 199–228.

E-Graph

- egg[5] - e-graphs implemented in Rust.

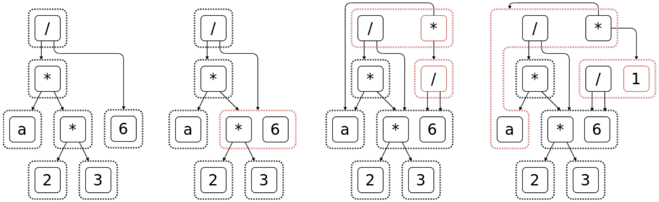

Figure 1: The four depicted e-graphs represent the process of equality saturation for the equivalent ways to write $a * (2 * 3)/6$. The dashed boxes represent equivalence classes, and regular boxes represent e-nodes.

[5] Max Willsey et al. "Egg: Fast and Extensible Equality Saturation". In: *Proc. ACM Program. Lang.* 5.POPL (Jan. 2021). DOI: 10.1145/3434304. URL: https://doi.org/10.1145/3434304.

The Path to the E-Graph

- Non-destructive rule application - copy everything
- Conceptually egraph is similar to fast data structure for sets of trees.
- Share subtrees if you can
- Share parents if you can
- Bipartite graph of ENodes and EClasses
- Union-Find and Hash Cons

Why Julia?

Julia[6]:

- is high-level, easy to learn
- is performant
- has excellent LISP-like metaprogramming and reflective capabilities
- has an excellent high-level dynamic type system
- supports generation and manipulation of programs as first-class values
- has an excellent package ecosystem for real world use cases

[6] Jeff Bezanson et al. "Julia: A fresh approach to numerical computing". In: *SIAM review* 59.1 (2017), pp. 65–98.

A Sketch of an E-Graph in Julia

```julia
struct ENode
    head::Symbol
    args::EClassId
end
struct EClass
    members::Set{ENode}
end
struct EGraph
    memo::Dict{ENode, EClassId}
    classes::Dict{EClassId,EClass}
    equiv::IntDisjointSet
end
```

Equality Saturation

Given a starting e-graph g, a set of rewrite rules t and an iteration limit n:

- For each rule in t, search through the e-graph for l.h.s.
- For each match produced, apply the rewrite
- Do a bottom-up traversal of the e-graph to rebuild the *congruence closure*
- If the e-graph hasn't changed from last iteration, it has **saturated**
- Loop at most n times.

Knowing if an expression with a set of rules saturates an e-graph or never terminates is still *an open research problem*.

What is so cool about e-graphs?

- When to apply which rewrite? *Phase ordering problem*
- Apply all rewrites simultaneously? Keep history? *Exponential space and time!*
- **Equality saturation** + **E-Graphs** = **E-Graph Rewriting**.
- $\implies$ no more phase ordering problem
- $\implies$ bidirectional equational rules for free!

With *e-graph rewriting* Here's an example rewrite system

```
using Metatheory
using Metatheory.Classic
@metatheory_init ()

th = @theory begin
    a + a => 2a     # this rule is ok!
    a + b => b + a  # causes loops in classical rewriting
    0 + a => a
end
```

Classical Rewriting vs E-Graph Rewriting

- Rule `a + b => b + a` causes an infinite rewrite loop
- User reasoning and rewriting strategies are required to make this terminate!

```julia
using Metatheory.Classic
expr = :(x + 0)
rewrite(expr, th)
```

Result will be `:(x + 0)`

Using *e-graph* rewriting with equality saturation, the problem is avoided.

```julia
using Metatheory.EGraphs
expr = :(x + 0)
g = EGraph(expr)
saturate!(g, th) # runs equality saturation
extract!(g, astsize) # extract the shortest expression from the e-graph g
```

Result will be `:x`

Symbolics.jl

Metatheory.jl can be used to rewrite Symbolics.jl expressions. This has proved really useful for optimization tasks: In[7] we developed a set of equality rules and a cost function to generate equivalent Symbolics.jl expressions which are computed in fewer CPU cycles. We tested against a 1122 ODE model of B Cell Antigen Signaling and simplified the 24388 terms using Symbolics.jl+Metatheory.jl. The generated code accelerated from 15.023μs to 7.372μs per execution, halving the time required to solve the highly stiff ODEs.

```
theory = @methodtheory begin
    a * x == x * a
    a * x + a * y == a*(x+y)
    -1 * a == -a
end
```

[7] Shashi Gowda et al. "High-performance symbolic-numerics via multiple dispatch". In: *arXiv preprint arXiv:2105.03949* (2021).

Symbolics.jl

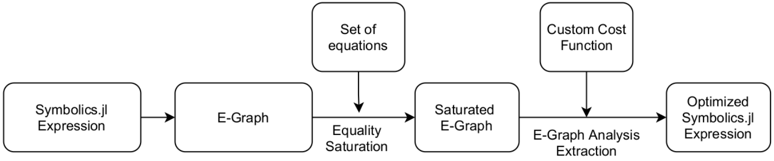

Figure 2: Pipeline of Symbolics.jl expression optimization

We are developing TermInterface.jl, a package for a shared interface that represents symbolic expressions. To rewrite on arbitrary expression types with Metatheory.jl, it will be necessary to implement a few methods from this interface (`isterm`, `arity`, `gethead`, `getargs`, `similarterm...`)

Fibonacci with SymbolicUtils.jl vs Metatheory.jl

SymbolicUtils.jl

```
@syms fib(x::Int)::Int

const rset = [
    @rule fib(0) => 0
    @rule fib(1) => 1
    @rule fib(~n) => fib(~n - 1) + fib(~n - 2)
] |> Chain |> Postwalk |> Fixpoint

compute_fib(n) = rset(fib(n))
```

Metatheory.jl

```
const fibo = @theory begin
    x::Int + y::Int |> x+y
    fib(n::Int) |> (n < 2 ? n : :(fib($(n-1)) + fib($(n-2))))
end;

function compute_fib(n)
    g = EGraph(:(fib($n)))
    saturate!(g, fibo, SaturationParams(timeout=7000, scheduler=SimpleScheduler))
    extract!(g, astsize)
end
```

vs. Fibonacci with Metatheory

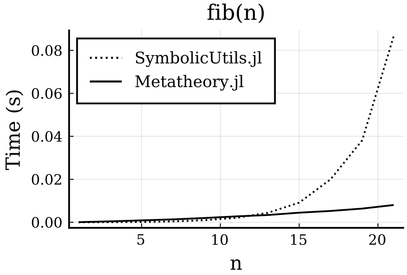

Figure 3: Benchmark of Fibonacci Sequence computation of SymbolicUtils.jl against Metatheory.jl. MT achieves implicit memoization of computation with e-graphs!

Category Theory

We have experimented with

- Categories - A powerful abstraction. LA, RA, functional programs can be modeled
- Rewriting Catlab.jl expressions using Metatheory.jl
- GATs[8]- Equational reasoning on types and terms
- Alternative algebraic encodings for GATs in pure Julia
- Proving equality of expressions over theories by e-graph rewriting

One possible encoding[9]:

- Guarded E-matching $(A \otimes B) \cdot (C \otimes D) = (A \cdot C) \otimes (B \cdot D)$
- Internalized type derivation

Is this the right way to go?

[8] Generalized Algebraic Theories
[9] https://www.philipzucker.com/metatheory-progress/

Going Faster than Knuth-Bendix

Hurwitz groups of type $(2, 3, 7, n)$. Is an expr. equal to ε? Thanks to Marek Kaluba (@kalmarek)

```
rep(x, op, n::Int) foldl((x, y) -> :(($op)$x, $y)), repeat([x], n))

T = [
    @rule a ⋆ :ε => a # identity element
    @rule :ε ⋆ a => a
    @rule a ⋆ (b ⋆ c) => (a ⋆ b) ⋆ c # associativity
    @rule (a ⋆ b) ⋆ c => a ⋆ (b ⋆ c)
    @rule :b⋆:B => :ε # inverses
    @rule :a⋆:a => :ε
    @rule :b⋆:b⋆:b => :ε
    @rule :B ⋆ :B => :B
    RewriteRule(Pattern(rep(:(:a⋆:b), :⋆, 7)), Pattern(:(:ε)))
    RewriteRule(Pattern(rep(:(:a⋆:b⋆:a⋆:B), :⋆, 12)), Pattern(:(:ε)))
]


g = EGraph(:(a⋆b⋆ a⋆a⋆a ⋆ b⋆b⋆b ⋆ a ⋆ B⋆B⋆B⋆B ⋆ a))
params = SaturationParams(timeout=8, scheduler=BackoffScheduler)
@timev saturate!(g, G, params)
@test extract!(g, astsize) == :ε
```

https://www.heldermann-verlag.de/gcc/gcc02/gcc028.pdf
Metatheory.jl takes around 0.04364s, KnuthBendix.jl fails!

Other Experiments

You can find other experiments in the `tests/` folder in the Metatheory.jl source code!

- λ-calculus partial evaluator
- Propositional logic/SAT solver
- WHILE-language interpreter/superinterpreter with reversible computations

Future directions

Internals:

- A shared interface package for interoperability between symbolic packages
- Better pattern matching
- Smarter scheduler algorithms for fast, more efficient saturation.
- Defining and implementing and algorithm for human-readable (but verifiable) proof paths in e-graphs (Oliver Flatt has been working on this)
- Better debugging and logging utilities. (What caused this failure? Which rules are problematic in a system?)

Experiments and Use Cases:

- Array Symbolics with Symbolics.jl!
- Optimizing the Julia IRs
- Optimizing Flux.jl deep learning models
- Optimizing quantum circuits for Yao.jl (Thanks to Chen Zhao)

Thank You!