Metatheory.jl: Fast and Elegant Algebraic Computation in Julia with Extensible Equality Saturation

Alessandro Cheli[1, 2]

1 Undergraduate Student, University of Pisa **2** Independent Researcher

DOI: 10.21105/joss.03078

Software

- Review
- Repository
- Archive

Editor: David P. Sanders

Reviewers:

- @mwillsey
- @jpfairbanks
- @philzook58

Submitted: 12 February 2021
Published: 29 March 2021

Statement of Need

The Julia programming language is a fresh approach to technical computing (Bezanson et al., 2017), disrupting the popular conviction that a programming language cannot be high-level, easy to learn, and performant at the same time. One of the most practical features of Julia is the excellent metaprogramming and macro system, allowing for *homoiconicity*: programmatic generation and manipulation of expressions as first-class values, a well-known paradigm found in LISP dialects such as Scheme.

Metatheory.jl is a general-purpose metaprogramming and algebraic computation library for the Julia programming language, designed to take advantage of its powerful reflection capabilities to bridge the gap between symbolic mathematics, abstract interpretation, equational reasoning, optimization, composable compiler transforms, and advanced homoiconic pattern-matching features. Intuitively, Metatheory.jl transforms Julia expressions into other Julia expressions at both compile time and run time. This allows users to perform customized and composable compiler optimizations that are specifically tailored to single, arbitrary Julia packages. The library provides a simple, algebraically composable interface to help scientists to implement and reason about all kinds of formal systems, by defining concise rewriting rules as syntactically-valid Julia code. The primary benefit of using Metatheory.jl is the algebraic nature of the specification of the rewriting system. Composable blocks of rewrite rules bear a strong resemblance to algebraic structures encountered in everyday scientific literature.

Summary

Metatheory.jl offers a concise macro system to define *theories*: composable blocks of rewriting rules that can be executed through two, highly composable, rewriting backends. The first is based on standard rewriting, built on top of the pattern matcher developed in (Zhao & Carlsson, 2020). This approach, however, suffers from the usual problems of rewriting systems. For example, even trivial equational rules such as commutativity may lead to non-terminating systems and thus need to be adjusted by some sort of structuring or rewriting order, which is known to require extensive user reasoning.

The other back-end for Metatheory.jl, the core of our contribution, is designed so that it does not require the user to reason about rewriting order. To do so it relies on equality saturation on *e-graphs*, the state-of-the-art technique adapted from the egg Rust library (Willsey et al., 2021).

E-graphs can compactly represent many equivalent expressions and programs. Provided with a theory of rewriting rules, defined in pure Julia, the *equality saturation* process iteratively executes an e-graph-specific pattern matcher and inserts the matched substitutions. Since e-graphs can contain loops, infinite derivations can be represented compactly and it is not required that the described rewrite system be terminating or confluent.

The saturation process relies on the definition of e-graphs to include *rebuilding*, i.e. the automatic process of propagation and maintenance of congruence closures. One of the core contributions of (Willsey et al., 2021) is a delayed e-graph rebuilding process that is executed at the end of each saturation step, whereas previous definitions of e-graphs in the literature included rebuilding after every rewrite operation. Provided with *equality saturation*, users can efficiently derive (and analyze) all possible equivalent expressions contained in an e-graph. The saturation process can be required to stop prematurely as soon as chosen properties about the e-graph and its expressions are proved. This latter back-end based on *e-graphs* is suitable for partial evaluators, symbolic mathematics, static analysis, theorem proving and superoptimizers.

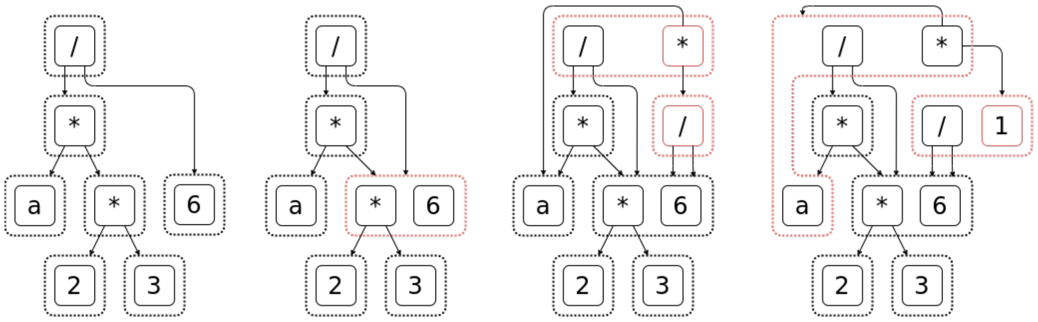

Figure 1: These four e-graphs represent the process of equality saturation, adding many equivalent ways to write $a * (2 * 3)/6$ after each iteration.

The original egg library (Willsey et al., 2021) is the first implementation of generic and extensible e-graphs (Nelson & Oppen, 1980); the contributions of egg include novel amortized algorithms for fast and efficient equivalence saturation and analysis. Differently from the original Rust implementation of egg, which handles expressions defined as Rust strings and data structures, our system directly manipulates homoiconic Julia expressions, and can therefore fully leverage the Julia subtyping mechanism (Zappa Nardelli et al., 2018), allowing programmers to build expressions containing not only symbols but all kinds of Julia values. This permits rewriting and analyses to be efficiently based on runtime data contained in expressions. Most importantly, users can – and are encouraged to – include type assertions in the left-hand side of rewriting rules in theories.

One of the project goals of Metatheory.jl, beyond being easy to use and composable, is to be fast and efficient. Both the first-class pattern matching system and the generation of e-graph analyses from theories rely on RuntimeGeneratedFunctions.jl (Rackauckas & Foster, 2021), generating callable functions at runtime that efficiently bypass Julia's world age problem (Belyakova et al., 2020) with the full performance of a standard Julia anonymous function.

Analyses and Extraction

With Metatheory.jl, modeling analyses and conditional/dynamic rewrites are straightforward. It is possible to check conditions on runtime values or to read and write from external data structures during rewriting. The analysis mechanism described in egg (Willsey et al., 2021) and re-implemented in our contribution lets users define ways to compute additional analysis metadata from an arbitrary semi-lattice domain, such as costs of nodes or logical statements attached to terms. Other than for inspection, analysis data can be used to modify expressions in the e-graph both during rewriting steps and after e-graph saturation.

Therefore using the equality saturation (e-graph) backend, extraction can be performed as an on-the-fly e-graph analysis or after saturation. Users can define their own cost function, or choose between a variety of predefined cost functions for automatically extracting the best-fitting expressions from an equivalence class represented in an e-graph.

Example Usage

In this example, we build rewrite systems, called `theories` in Metatheory.jl, for simplifying expressions in the usual commutative monoid of multiplication and the commutative group of addition, and we compose the `theories` together with a *constant folding* theory. The pattern matcher for the e-graphs backend allows us to use the existing Julia type hierarchy for integers and floating-point numbers with a high level of abstraction. As a contribution over the original egg (Willsey et al., 2021) implementation, left-hand sides of rules in Metatheory.jl can contain type assertions on pattern variables, to give rules that depend on consistent type hierarchies and to seamlessly access literal Julia values in the right-hand side of dynamic rules.

We finally introduce two simple rules for simplifying fractions, that for the sake of simplicity, do not check any additional analysis data. Figure 1 contains a friendly visualization of a consistent fragment of the equality saturation process in this example. You can see how loops evidently appear in the definition of the rewriting rules. While the classic rewriting backend would loop indefinitely or stop early when repeatedly matching these rules, the e-graph backend natively supports this level of abstraction and allows the programmer to completely forget about the ordering and looping of rules. Efficient scheduling heuristics are applied automatically to prevent instantaneous combinatorial explosion of the e-graph, thus preventing substantial slowdown of the equality saturation process.

```julia
using Metatheory
using Metatheory.EGraphs

comm_monoid = @theory begin
  # commutativity
  a * b => b * a
  # identity
  a * 1 => a
  # associativity
  a * (b * c) => (a * b) * c
  (a * b) * c => a * (b * c)
end;

comm_group = @theory begin
  # commutativity
  a + b => b + a
  # identity
  a + 0 => a
  # associativity
  a + (b + c) => (a + b) + c
  (a + b) + c => a + (b + c)
  # inverse
  a + (-a) => 0
end;

# dynamic rules are defined with the `|>` operator
folder = @theory begin
  a::Real + b::Real |> a+b
  a::Real * b::Real |> a*b
end;

div_sim = @theory begin
  (a * b) / c => a * (b / c)
  a::Real / a::Real  |>  (a != 0 ? 1 : error("division by 0"))
end;

t = union(comm_monoid, comm_group, folder, div_sim) ;

g = EGraph(:(a * (2*3) / 6)) ;
saturate!(g, t) ;
ex = extract!(g, astsize)
# :a
```

Conclusion

Many applications of equality saturation to advanced optimization tasks have been recently published. Herbie (Panchekha et al., 2015) is a tool for automatically improving the precision of floating point expressions, which recently switched to egg as the core rewriting backend. However, Herbie requires interoperation and conversion of expressions between different languages and libraries. In (Yang et al., 2021), the authors used egg to superoptimize tensor signal flow graphs describing neural networks. Implementing similar case studies in pure Julia would make valid research contributions on their own. We are confident that a well-integrated and homoiconic equality saturation engine in pure Julia will permit exploration of many new metaprogramming applications, and allow them to be implemented in an elegant, performant and concise way. Code for Metatheory.jl is available in (Cheli, 2021), or at https://github.com/0x0f0f0f/Metatheory.jl.

Acknowledgements

We acknowledge Max Willsey and contributors for their work on the original egg library (Willsey et al., 2021), Christopher Rackauckas and Christopher Foster for their efforts in developing RuntimeGeneratedFunctions (Rackauckas & Foster, 2021), Taine Zhao for developing MLStyle (Zhao, 2021) and MatchCore (Zhao & Carlsson, 2020), and Philip Zucker for his original idea of implementing E-Graphs in Julia (Zucker, 2020b, 2020a) and support during the development of the project. Special thanks to Filippo Bonchi for a friendly review of a preliminary version of this article.

References

Belyakova, J., Chung, B., Gelinas, J., Nash, J., Tate, R., & Vitek, J. (2020). World age in julia: Optimizing method dispatch in the presence of eval. *Proc. ACM Program. Lang.*, *4*(OOPSLA). https://doi.org/10.1145/3428275

Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, *59*(1), 65–98. https://doi.org/10.1137/141000671

Cheli, A. (2021). Metatheory.jl. In *GitHub repository*. GitHub. https://github.com/0x0f0f0f/Metatheory.jl

Nelson, G., & Oppen, D. C. (1980). Fast decision procedures based on congruence closure. *J. ACM*, *27*(2), 356–364. https://doi.org/10.1145/322186.322198

Panchekha, P., Sanchez-Stern, A., Wilcox, J. R., & Tatlock, Z. (2015). Automatically improving accuracy for floating point expressions. *SIGPLAN Not.*, *50*(6), 1–11. https://doi.org/10.1145/2813885.2737959

Rackauckas, C., & Foster, C. (2021). RuntimeGeneratedFunctions.jl: Functions generated at runtime without world-age issues. In *GitHub repository*. GitHub. https://github.com/SciML/RuntimeGeneratedFunctions.jl

Willsey, M., Nandi, C., Wang, Y. R., Flatt, O., Tatlock, Z., & Panchekha, P. (2021). Egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, *5*(POPL). https://doi.org/10.1145/3434304

Yang, Y., Phothilimtha, P. M., Wang, Y. R., Willsey, M., Roy, S., & Pienaar, J. (2021). Equality saturation for tensor graph superoptimization. *arXiv Preprint arXiv:2101.01332*. https://arxiv.org/pdf/2101.01332.pdf

Zappa Nardelli, F., Belyakova, J., Pelenitsyn, A., Chung, B., Bezanson, J., & Vitek, J. (2018). Julia subtyping: A rational reconstruction. *Proc. ACM Program. Lang.*, *2*(OOPSLA). https://doi.org/10.1145/3276483

Zhao, T. (2021). MLStyle.jl: Fast, consistent and extensible functional programming infrastructures. In *GitHub repository*. GitHub. https://github.com/thautwarm/MLStyle.jl

Zhao, T., & Carlsson, K. (2020). MatchCore.jl: A minimal implementation of optimized pattern matching. In *GitHub repository*. GitHub. https://github.com/thautwarm/MatchCore.jl

Zucker, P. (2020a). *E-graph pattern matching Part II*. https://www.philipzucker.com/egraph-2/

Zucker, P. (2020b). *E-graphs in julia Part I*. https://www.philipzucker.com/egraph-1/