An Interactive Visualizer for Equality Solvers using E-Graphs
Princeton University COS 516 Course Project (Fall '22)

Ryan Torok
Princeton University

Leon Schürmann
Princeton University

1 Introduction

The *Theory of Equality* (TOE) is a theory in first-order logic which axiomatizes that the equality operator (=) behaves reasonably [2]. A well-known algorithm for finding all equivalence classes of a given formula, which constitutes a theory solver for TOE, is to list out all terms that appear in the formula, then iterate through pairs of terms, adding terms to equivalence classes using a union-find data structure by repeatedly applying the rule $a = b \rightarrow f(a) = f(b)$, where a and b are terms, and f is a function symbol. In the worst case, running this procedure until no more matches are possible requires quadratic time. While this is still a polynomial-time algorithm, it is rather naive in its method to choose new pairs and it is definitely possible to do better.

E-graphs [3] improve the time complexity of this procedure by representing the terms of a formula as a directed acyclic graph (DAG), such that each constant symbol or function symbol is given a single node, and the composition of terms is represented by the edges. For example, Figure 1 represents an e-graph for the formula $(a*2)/2$. The structure of these graphs make it possible to efficiently find all equality classes without requiring the linear pass over all terms, and also make it possible to support an arbitrary set of additional rewrite rules. `egg` is a popular Rust library that implements e-graphs, and has seen frequent use in implementations of automated verification programs, largely due to its flexibility [1].

Unfortunately, the precise mechanisms employed both within the congruence closure algorithm, as well as the extended e-graph data structures, can be unintuitive for novice users. We presume that an interactive visualization can help demonstrate the basic concepts behind reasoning about equality under a given set of rewrite rules. In this project, we aim to create a web-based visualization tool that allows the user to interactively explore the equivalence-matching procedure and program optimization techniques in a step-by-step manner.

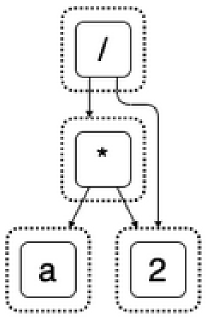
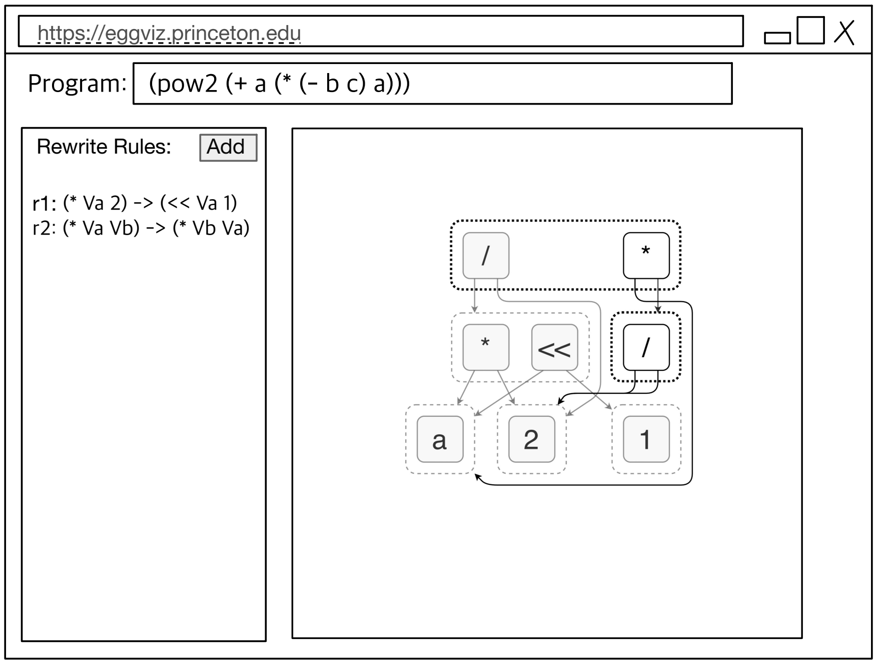


Figure 1: E-graph representing the formula $(a*2)/2$

2 Interface Design

Figure 2 shows a high-level layout of our proposed interface. Our web application will consist of an expression field, a left pane containing the set of user-defined rewrite rules, and a main view containing the e-graph. For the graph visualization, we plan to use the `vis` graph visualization library that was also used in the CDCL solver web application. The displayed graph will also indicate the found equivalence classes using either a dotted-line box (as in the `egg` paper), or using color coordination. As the user steps through the reduction, the visualization will indicate which equality classes are at each step and the responsible rewrite rule. We may also choose to support "tentative changes", where the user can hover over one of the classes in a possible merge and observe the effect without committing it.

The expression field will accept a program using prefix notation in a LISP-like syntax. For example, the program `pow2(a + (b * c))` would be written as `(pow2 (+ a (* b c)))`. Requiring the user to enter programs in this format makes them simple to parse and has the particular advantage that the arity of each function symbol is fully deterministic and does not need to be manually specified by the user. The rewrite rules box will accept the same syntax,



Program: (pow2 (+ a (* (- b c) a)))

Rewrite Rules:

r1: (* Va 2) -> (<< Va 1)
r2: (* Va Vb) -> (* Vb Va)

Figure 2: Illustration of our proposed web interface.

as well as the -> operator to represent "replaces".

3 Implementation Strategy

The system shall be implemented as a standalone web-application capable of running in modern Web browsers with WebAssembly support. It does not require any backend program and executes as a self-contained program in the user's web-browser.

In accordance with the requirements posed by the interface design section, the project shall implement a basic user interface in HTML5, CSS and JavaScript, possibly using pre-existing user-interface and frontend application frameworks. This user interface incorporates a library for visualizing (intermediate) e-graphs maintained by the underlying application logic.

To actually represent the e-graph structure, as well as to perform any of the supported transformations and other operations (such as a step-by-step saturation of the e-graph), the system employs the `egg` library written in Rust. This library is extended with the necessary logic to interoperate with the frontend written in JavaScript, and embedded into the system by means of compiling it to a WebAssembly library.

In order to integrate with our step-by-step visualization, we plan to transform `egg`'s equality saturation algorithm to allow our application read the intermediate state after each iteration of its loop, which we can use to produce a visualization of the application of rewrite rules.

4 Possible Extensions

It may be feasible to extend the application to not only operate on `egg`'s chosen order of rewrite rule applications, but to instead allow the user to guide applications of individual rewrite rules between terms of equivalence classes.

5 Project Schedule

We plan to build the proposed application by following roughly these steps:

1. Build the basic project scaffold with a rudimentary frontend and an `egg` dummy application (*1 week*).
2. Implement parsing of the entered program (automatically deducing function & constant symbols and creating appropriate Rust types to integrate with `egg`), as well as parsing of the rewrite rules (*1 week*).
3. Change `egg`'s implemented equivalence saturation algorithm to provide a step-by-step interface and return additional context for integration of our visualization (*1-2 weeks*).
4. Implement a JavaScript-based visualization for (intermediate) e-graphs. This likely includes a routine to walk

two e-graphs and create a *diff* to update a pre-existing rendered e-graph (*1-2 weeks*).

5. Finish and debug the implementation; write the project report (*2 weeks*).

References

[1] egg: e-graphs good. https://egraphs-good.github.io/.

[2] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation*. Springer, New York, 2007.

[3] Greg Nelson. Techniques for program verification. 1981.