E-Graphs, VSAs, and Tree Automata: a Rosetta Stone

Yisu Remy Wang*
University of Washington
remywang@cs.washington.edu

James Koppel
MIT
jkoppel@mit.edu

Altan Haan
OctoML
ahaan@octoml.ai

Josh Pollock
MIT
jopo@mit.edu

Many tasks in programming languages involve representing and manipulating sets of programs. In program synthesis, the goal is to find a program satisfying a given specification from a set of programs possibly generated by a given grammar. In program optimization, the goal is to find an efficient program from the set of programs equivalent to the input. Programming languages research has considered various abstractions to represent sets of programs. Two examples are the version space algebra (VSA) [Lau et al. 2003; Mitchell 1982], popularized by FlashFill [Gulwani 2011] for enumerative program synthesis, and the e-graph [Nelson 1980; Nieuwenhuis and Oliveras 2005] that lies at the heart of an array of new program optimizers [Willsey et al. 2021]. In this talk we show that VSAs and e-graphs are but special cases of the well-studied finite-state (tree) automata from formal language theory. This new perspective allows us to place VSAs and e-graphs on a firm theoretical foundation, and also enables us to leverage powerful tools from formal language theory to perform tasks in programming languages. In the converse, bridging the concepts can also contribute to tree automata research with techniques developed for e-graphs and VSAs.

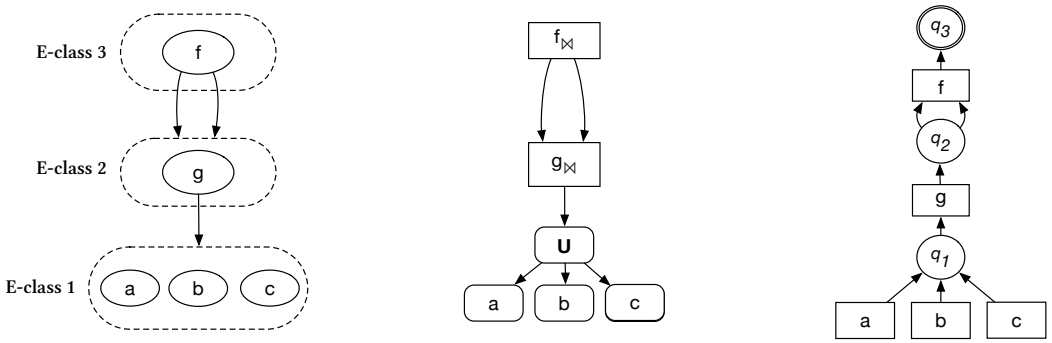
Background. For an intuition of e-graphs, VSAs and tree automata we give an example of each. Consider representing the set of 9 terms $\mathcal{T} = \{f(g(X), g(Y))\}$, where $X, Y \in \{a, b, c\}$. Fig. 1 gives the e-graph, VSA, and tree automaton representing this set. In an e-graph, alternative expressions are grouped together to form the so-called e-classes, shown in dotted boxes in Fig. 1a. The expressions a, b and c are grouped together under e-class 1. Function symbols then point to these groups as children, while aggressively sharing common sub-parts. For example, f points to e-class 2 twice as its left and right children. In VSA, alternatives are combined with the union operator $\mathbf{U}$, while function symbols may point to such union operators. Finally, in a tree automaton multiple expressions may lead to the same state, and each function symbol connects its children states to a parent state. The similarity between the three is striking, and they all appear to function by grouping and sharing subterms in the same way. Though works involving VSAs, e-graphs, and tree automata typically construct and use them very differently, their representational capabilities appear quite similar.

In fact, we shall see that tree automata strictly generalize e-graphs and VSAs. In particular, when we impose different invariants on a tree automaton, we immediately obtain an e-graph or a VSA as special cases. By taking an automata-theoretic perspective, we benefit from the long and fruitful history of research on finite state automata. We are empowered to leverage its powerful tools for tasks in programming languages. We share an early result from this synergy, by improving e-graph-based optimizers using automata minimization algorithms. We will also discuss several avenues for future work that study the connection between the three, including applications in invariant inference and termination analysis by completion.

Automata Minimization. A classic operation on finite state automata is minimization. Given a automaton, minimization returns an automaton that recognizes the same language but contains fewer states. Because an e-graph is a tree automaton, we can use the same minimization procedure to reduce the size of any e-graph. A smaller e-graph can lead to faster optimization in equality saturation, because the extraction phase of equality saturation runs in time dependent on the e-graph size. For example, many applications of equality saturation aim to eliminate common subexpressions [Wang et al. 2020; Yang et al. 2021]. Most of them achieve this goal by encoding the extraction problem into an integer linear program (ILP). ILP-solving is NP-hard in general, and has been shown to be expensive in practical applications of equality saturation. Several heuristics have been proposed to sacrifice optimality for faster extraction [Yang et al. 2021]. In contrast, we show that reducing the e-graph size by minimization can shorten extraction time while guaranteeing optimality.

We implemented a minimization algorithm in the egg [Willsey et al. 2021] library for equality saturation, and ran experiments using the library's integration test suites. One test suite implements standard algebraic rules on simple arithmetic, as well as some elementary rules for symbolic differentiation; the other contains rewrite rules for a small language based on lambda calculus. We run equality saturation on all input expressions found in each test suite and optimize for expression size. In particular, we compute size after reusing

*Wang is supported by NSF IIS 1907997 and NSF IIS 1954222.



(a) E-graph. **(b)** Version-space algebra. **(c)** Tree automaton.

Figure 1. Comparison of an e-graph, a VSA, and a tree automaton.

common subexpressions, so the expression $x + x$ has a size of 2 (1 for + and 1 for x) instead of 3. To extract the optimal program, we follow the ILP encoding in [Tate et al. 2009] and assign a cost of 1 to every e-node. The total cost of a program is therefore exactly the number of e-nodes it contains.

To investigate the impact of minimization, we measure the extraction time before and after minimization, as we show in Figure 2. We see that minimization can lead to faster extraction especially for larger instances.

Naturally, one may ask what kind of classes and expressions got merged during minimization. Surprisingly, when we sampled pairs of terms from merged classes, virtually every pair could have been proven equivalent using the rewrite rules! Then why were they not merged during rewrite application? The reason is that missed equivalences are not so unusual in practical applications of equality saturation. Because most application domains require rules that cause explosive, or even unbounded growth of the e-graph, rewrite application usually stops before reaching a fix point. This means if one were to apply more rewrites, it is possible to identify additional equivalences in the e-graph. This observation leads to an alternative way to shrink the e-graph by using the rewrite rules: simply perform a second phase of rewrite application, but only apply a rewrite if it combines existing classes without introducing new nodes. Because this second phase only combines existing classes which makes the e-graph smaller, it will reach a fix point in time at most linear to the e-graph size. Using rewriting rules may also result in an e-graph smaller than one obtained with automata minimization, because the rules provide semantic information whereas an automaton treats its terms purely syntactically. We can also compose the two approaches to further reduce the e-graph size. Specifically, we invoke the automata minimization algorithm after the second phase of merge-only rewrite applications. Figure 2 compares the impact on extraction time of automata minimization, merge-only rewrites, and a combination of both. Depending on the situation, any of the three methods may lead to the fastest extraction.

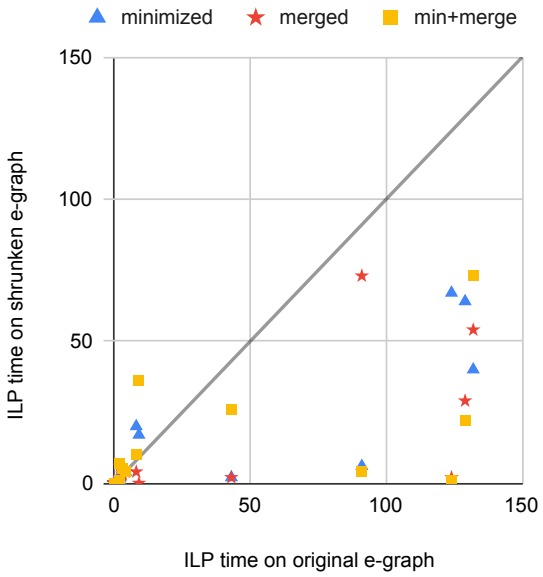


Figure 2. Impact of minimization on ILP time.

Using E-graphs to Prove Unreachability. We have also discovered that the tree automata community has studied an operation identical to that of extending an e-graph with a rewrite rule, namely the *tree automata completion* of Genet et al [Feuillade et al. 2004; Genet 2014, 2016; Genet and Rusu 2010]. From this connection, we are able to port ideas from this line of work to e-graphs. In particular, Genet et al use the idea of *overapproximating* tree automata completion by adding extra edges between nodes (eclasses). When overapproximated completion reaches saturation, the resulting tree automaton soundly overapproximates the set of terms reachable via rewrites. It would be straightforward to use this idea in e-graphs, allowing them to be used to prove unreachability and thus to prove safety properties.

References

Guillaume Feuillade, Thomas Genet, and Valérie Viet Triem Tong. 2004. Reachability Analysis over Term Rewriting Systems. *Journal of Automated Reasoning* 33, 3 (2004), 341–383.

Thomas Genet. 2014. *A Note on the Precision of the Tree Automata Completion.* Ph.D. Dissertation. IRISA.

Thomas Genet. 2016. Termination Criteria for Tree Automata Completion. *Journal of Logical and Algebraic Methods in Programming* 85, 1 (2016), 3–33.

Thomas Genet and Vlad Rusu. 2010. Equational Approximations for Tree Automata Completion. *Journal of Symbolic Computation* 45, 5 (2010), 574–597.

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets using Input-Output Examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330.

Tessa Lau, Steven A Wolfman, Pedro Domingos, and Daniel S Weld. 2003. Programming by Demonstration Using Version Space Algebra. *Machine Learning* 53, 1 (2003), 111–156.

Tom M. Mitchell. 1982. Generalization as Search. *Artif. Intell.* 18, 2 (1982), 203–226. https://doi.org/10.1016/0004-3702(82)90040-6

Charles Gregory Nelson. 1980. *Techniques for program verification.* Stanford University.

Robert Nieuwenhuis and Albert Oliveras. 2005. Proof-Producing Congruence Closure. In *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3467)*, Jürgen Giesl (Ed.). Springer, 453–468. https://doi.org/10.1007/978-3-540-32033-3_33

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 264–276.

Yisu Remy Wang, Shana Hutchison, Dan Suciu, Bill Howe, and Jonathan Leang. 2020. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *Proc. VLDB Endow.* 13, 11 (2020), 1919–1932. http://www.vldb.org/pvldb/vol13/p1919-wang.pdf

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and Extensible Equality Saturation. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–29.

Yichen Yang, Mangpo Phitchaya Phothilimtha, Yisu Remy Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. *CoRR* abs/2101.01332 (2021). arXiv:2101.01332 https://arxiv.org/abs/2101.01332