Destructive E-Graph Rewrites

Paul Zhang
paulz@ucla.edu
University of California, Los Angeles
USA

Yisu Remy Wang
remywang@cs.ucla.edu
University of California, Los Angeles
USA

Abstract

Equality saturation is an automatic rewrite technique with applications ranging from theorem proving to program optimization. It relies on the e-graph data structure, which compactly represents many equivalent terms. A significant bottleneck in equality saturation is unbounded and exponential e-graph growth. Destructive e-graph rewrites present a promising mitigation. Any rewrite rule can be made destructive by e-matching the left hand side and removing matches corresponding to the right hand side, rather than applying those matches. Applying destructive rewrites in `egg`, a state-of-the-art library for equality saturation, demonstrates up to 3x speed-ups and solves benchmarks that otherwise exceed the e-node limit.

1 Introduction

Equality saturation relies on *non-destructive* term rewriting, which avoids the trap of local optima during optimization. To achieve that, the underlying e-graph data structure can compactly represent a large space of equivalent programs. Nevertheless, it is common for the e-graph to grow without bound, exceeding practical memory and computation limits. This phenomenon is colloquially known as "e-graph explosion" in the community. A variety of mitigations have been developed to address e-graph explosion:

1. In the `egg` library [6], unbounded e-graph growth is partially addressed through *schedulers*, which control when rewrites are applied. For example, the default `BackoffScheduler` temporarily bans any rewrite when it is applied too many times. This only delays e-graph explosions in hopes of giving other rewrites time to "catch up."
2. *Pruning* identifies e-nodes that can be removed without affecting the best term, but it requires domain-specific knowledge. For example, the e-nodes `(Num 0)` and `(Mul (Num 0) (Num 1))` belong to the same e-class, but the `Mul` e-node can be safely removed because `(Num 0)` is already the best way to represent `0`. Pruning is used by the `math.rs` tests in `egg`.
3. A *solver* can speculatively unify terms with which equality saturation would otherwise struggle to unify — for example, associativity and commutativity are prone to causing e-graph explosions [4]. However, this also requires domain-specific knowledge.
4. *Re-initializing* with the best extracted term (after a time limit) can forestall e-graph explosions at the cost of potentially missing valuable optimizations [2]. This represents a balance between the exhaustive nature of equality saturation and the speed of greedy rewriting. It also requires a good heuristic for determining the best term before re-initialization.
5. *Guided equality saturation* relies on terms or "sketches" that act as intermediate goals for a series of equality saturations [3]. Once the first equality saturation has reached the first intermediate goal, the second equality saturation is initialized with the best term from the first, and so on. The intermediate goals are meant to be supplied by human experts, so unfortunately it is not fully automatic.

Although these techniques remedy explosion to some degree, there is no complete solution due to the inherent exponential nature of the optimization problem. In this work, we propose destructive rewrites as yet another tool to help tame e-graph growth.

We demonstrate the main idea behind destructive rewrites with a simple example. Suppose the inital term to be optimized contains the following expression:

$$x_1 \times y_1 \times y_2 \times \cdots \times y_n + y_n \times y_{n-1} \times \cdots \times y_1 \times x_2$$

To factor out $y_1 \times \cdots \times y_n$, one would have to apply a long chain of associativity and commutativity to find the correct permutation, which would in term enable the distributive rule. Doing so will insert a large number of terms into the e-graph, but these terms are unlikely to be useful for further rewrites. Such terms may therefore be "garbage collected" once we have successfully applied distributivity. We believe such patterns to arise naturally in both theorem proving and optimization: once certain rewrites "serve their purpose," we can safely discard the intermediate terms they created.

2 Making Rewrites Destructive

Listing 1 describes how to apply a rewrite destructively. Note that an e-graph is grounded if at least one term can be extracted from it. Removing an e-node is only safe if the resulting e-graph is grounded.

EGRAPHS '25, Seoul, SK,
2025. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```python
def apply_destructively(rewrite, egraph):
    matches = rewrite.lhs.search(egraph)

    for match in matches:
        remove_top_enode(rewrite.rhs, match, egraph)

    remove_unreachable(egraph)

def remove_top_enode(rhs, match, egraph):
    if (enode := rhs[match]) is not ENode:
        # Sometimes the RHS is an e-class
        # instead of an e-node
        return

    eclass = egraph.lookup(enode)

    if not grounded(eclass, enode, egraph):
        return

    eclass.nodes.remove(enode)
    egraph.hashcons.remove(enode)

visited = {}
def grounded(eclass, excluded, egraph):
    if eclass in visited:
        return False

    without_excluded = eclass.nodes.remove(excluded)

    for enode in without_excluded:
        if enode.is_leaf():
            return True

    for enode in without_excluded:
        if enode.children.all(
            lambda eclass:
                grounded(eclass, excluded, egraph)
        )

def remove_unreachable(egraph):
    for id in egraph.eclass_map:
        if not reachable(id, egraph.roots):
            egraph.eclass_map.remove(id)

    for enode in egraph.hashcons:
        if not reachable(enode, egraph.roots):
            egraph.hashcons.remove(enode)
```

Listing 1. Pseudocode for applying a rewrite destructively along with supporting methods.

2.1 When to Apply Destructive Rewrites

Choosing when to apply destructive rewrites is crucial for their practical use. Applying too often results in losing too much information, while applying too little is ineffective in mitigating e-graph explosions.

One strategy is to monitor how often every rewrite is applied. Any rewrite exceeding a certain threshold is applied destructively, after which the threshold is multiplicatively raised for that rewrite. This is similar to how `BackoffScheduler` works in egg.

More sophisticated strategies could choose to only destructively apply rewrites which do not immediately result in lower-cost terms, such as associativity and commutativity.

3 Case Study: egg

The strategy mentioned previously is simple, yet surprisingly effective. We benchmarked this strategy in the egg library's `math.rs` test suite. For the baseline configuration, we disabled both explanations [1] and pruning. Explanations are not supported by our current implementation of destructive rewrites, so it represents overhead that we should avoid measuring. And since we want to compare against equality saturation without domain-specific explosion mitigations, pruning is disabled. `BackoffScheduler` is still enabled, which is the default scheduler in egg.

The results are summarized in Figure 1. Although the fastest displayed speed-up is 3x in the `math_associate_adds` test, two of the tests (`diff_power_harder` and `integ_part2`) fail on baseline due to reaching the one-million node limit. But with destructive rewrites each test completes in under 0.03 seconds, suggesting that destructive rewrites could be even faster on larger e-graphs.

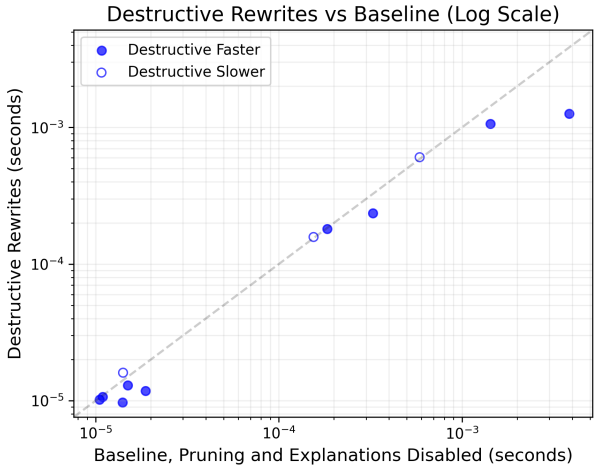

Figure 1. Log-scaled scatterplot comparing runtime of destructive rewrites against baseline configuration in the `math.rs` tests in egg. Baseline disables explanations, which our current implementation of destructive rewrites does not support, and disables pruning so there are no domain-specific mitigations against e-graph explosions (`BackoffScheduler`, which works for any e-graph, is still enabled).

4 Future work

We are currently benchmarking destructive rewrites in Herbie, a research tool for optimizing the accuracy of floating point expressions [5]. As discussed in Section 2.1, we will experiment with different strategies of applying destructive rewrites to improve performance. Support for explanations is also an important next step, since it is a commonly used feature in projects using egg. Furthermore, explanations use a separate e-graph within egg that contains more information, which could be exploited to enable more fine-grained control over destructive rewrites.

Acknowledgments

We are grateful to Pavel Panchekha for helping us run destructive rewrites on Herbie and to Oliver Flatt for valuable explanations of explanations.

References

[1] [n. d.]. https://docs.rs/egg/latest/egg/tutorials/_03_explanations/index.html

[2] Smail Kourta, Adel Abderahmane Namani, Fatima Benbouzid-Si Tayeb, Kim Hazelwood, Chris Cummins, Hugh Leather, and Riyadh Baghdadi. 2022. Caviar: an e-graph based TRS for automatic code optimization. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction* (Seoul, South Korea) *(CC 2022)*. Association for Computing Machinery, New York, NY, USA, 54–64. https://doi.org/10.1145/3497776.3517781

[3] Thomas Kundefinedhler, Andrés Goens, Siddharth Bhat, Tobias Grosser, Phil Trinder, and Michel Steuwer. 2024. Guided Equality Saturation. *Proc. ACM Program. Lang.* 8, POPL, Article 58 (Jan. 2024), 32 pages. https://doi.org/10.1145/3632900

[4] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing structured CAD models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 31–44. https://doi.org/10.1145/3385412.3386012

[5] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically improving accuracy for floating point expressions. *SIGPLAN Not.* 50, 6 (June 2015), 1–11. https://doi.org/10.1145/2813885.2737959

[6] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and Extensible Equality Saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434304