Bindings in e-graphs

EMMANUEL ANAYA-GONZALEZ, COLE KURASHIGE[*], UC San Diego, USA
ADITYA GIRIDHARAN, UC San Diego, USA
NADIA POLIKARPOVA, UC San Diego, USA

Applications of e-graphs to program synthesis have recently seen a rise in popularity. The efficient equality saturation workload that they enable makes them an ideal tool for program search and optimization. Unfortunately, straightforward approaches to dealing with binders such as those introduced by anonymous functions cause equality saturation to be overrun with additional terms, making the process slow and often intractable. Yet many of us would still like to have binders in the languages on which we perform equality saturation. We give an overview of approaches to dealing with binders and propose a fast and simple method that keeps the e-graph size down by culling unnecessary terms.

1 INTRODUCTION

To illustrate the problems we address in this paper, we consider the lambda calculus evaluator presented in Section 5 of [Willsey et al. 2021].

1.1 Beta-reduction in e-graphs

In a lambda calculus defined over an e-graph, instead of having terms like $\lambda x.\, 1$, you have e-nodes like $\lambda x.\, e$[1] where e could refer to any number of equivalent e-nodes, such as 1, $1 + 0$, or $(\lambda x.\, x)\ 1$. These e-nodes may even refer to their containing e-class e, forming a cycle that represents an infinite number of terms!

So while for a regular lambda calculus it's simple to say what $(\lambda x.\, 1)$ `false` is, it's generally difficult (and ill-advised) to directly compute $(\lambda x.\, e)$ `false` in an e-graph.

A straightforward workaround is to make substitution explicit as in Figure 1. These rewrite rules correspond to a standard small-step semantics for substitution. Instead of traversing the e-graph and generating new terms with x replaced with `false`, we beta-reduce $(\lambda x.\, e)$ `false` by rewriting it to $e[x \mapsto \texttt{false}]$ and then letting rewrite rules match each possible combination of substitutions that can be made across the e-nodes in e. This is easy, but unfortunately inefficient.

1.2 An illustrative example

Our running example for this paper is the term

$$(\lambda x.\, (y + y) + x)\ 1 \tag{1}$$

Beta-reducing yields the term

$$(y + y) + 1 \tag{2}$$

Figure 2a depicts Equation 1 as an e-graph and Figure 3b depicts this e-graph with Equation 2 added to it. Ideally, saturating Figure 2a should transform it to Figure 3b directly. However, because

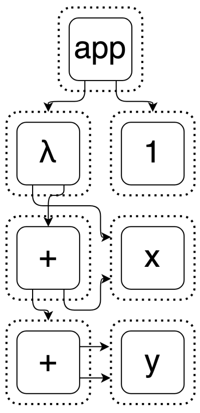
[*]Both authors contributed equally to this research.
[1]As a slight abuse of notation, when we write x in $\lambda x.\, e$, it refers to a singleton e-class containing the symbol x; the same is true of other constants like 1 or `false`.

Authors' addresses: Emmanuel Anaya-Gonzalez, Cole Kurashige, fanayagonzalez,ckurashige@ucsd.edu, UC San Diego, USA; Aditya Giridharan, agiridharan@ucsd.edu, UC San Diego, USA; Nadia Polikarpova, npolikarpova@ucsd.edu, UC San Diego, USA.

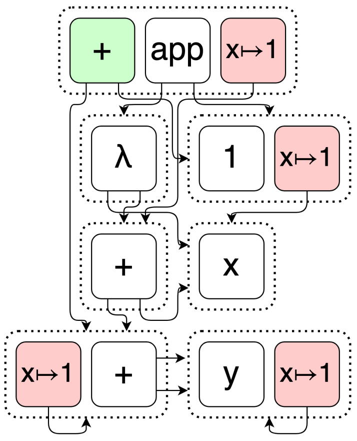
2023. 2475-1421/2023/4-ART1 $15.00
https://doi.org/

$$\begin{aligned}
(\lambda x.\, e_1)\; e_2 &\Rightarrow e_1[x \mapsto e_2] \\
(e_1\; e_2)[x \mapsto e] &\Rightarrow (e_1[x \mapsto e])\; (e_2[x \mapsto e]) \\
(\lambda x.\, e_1)[x \mapsto e_2] &\Rightarrow \lambda x.\, e_1 \\
(\lambda y.\, e_1)[x \mapsto e_2] &\Rightarrow \lambda y.\, (e_1[x \mapsto e_2]]) && \text{if } y \notin \mathrm{fv}(e_2) \\
(\lambda y.\, e_1)[x \mapsto e_2] &\Rightarrow \lambda \mathit{fresh}.\, (e_1[y \mapsto \mathit{fresh}][x \mapsto e_2]]) && \text{if } y \in \mathrm{fv}(e_2) \\
(e_1 + e_2)[x \mapsto e] &\Rightarrow e_1[x \mapsto e] + e_2[x \mapsto e] \\
x[x \mapsto e] &\Rightarrow e \\
y[x \mapsto e] &\Rightarrow y \\
\mathit{const}[x \mapsto e] &\Rightarrow \mathit{const}
\end{aligned}$$

Fig. 1. Small-step substitution rules. All e variables refer to e-classes. x and y refer to variables and fresh is a fresh variable name. const refers to a constant like 1 or `false`. fv(e) gives the set of free variables in e.



(a)



(b)

Fig. 2. 2a is the initial e-graph representing $(\lambda x.\,(y + y) + x)\; 1$. 2b is the e-graph run to saturation using small-step semantics for substitution. Even though we reach the goal $(y + y) + 1$ (green), we also end up with many intermediate substitution nodes (red).

beta-reduction is implemented via small-step substitution nodes, the actual e-graph that we get is Figure 2b.

For this simple example, the difference between Figure 2b and Figure 3b is small, but as the size of the e-graph increases, the additional substitution nodes add a significant (up to two times) overhead to e-graph size and therefore the time it takes to reach saturation.

2 PRIOR WORK

Many people who use e-graphs to rewrite programs try to avoid binders because of their overhead. For example, in [Smith et al. 2021], the authors note that adding binders incurred too much performance degradation to justify their utility. They sidestep these issues by designing their language without explicit binders.

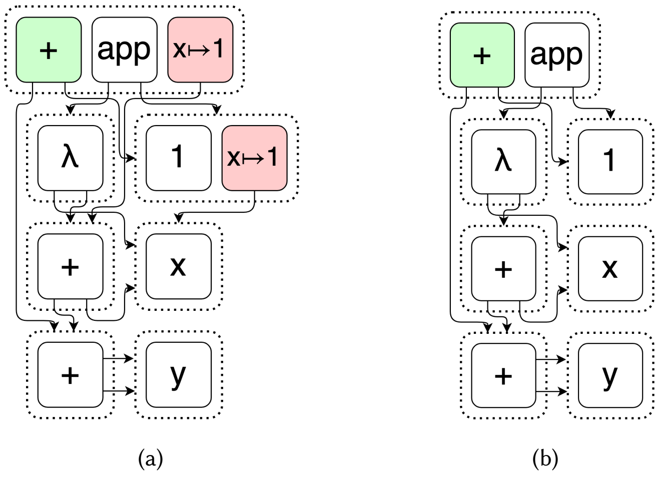


Fig. 3. 3a shows the saturated egraph when using the free variable check optimization described in 3.1. Note how the substitutions applied to y and $y + y$ are avoided. 3b shows the saturated egraph when using the free variable check and substitution node culling optimizations. The only node we end up adding represents our goal $(y + y) + 1$.

In [Koehler et al. 2022], the authors observe similar performance issues. To get around them, they introduce a faster, but incomplete approach. When rewriting a beta-reduction like $(\lambda x.\, e_1)\ e_2$, the authors extract from the e-graph the best term t_1 from e_1 and the best term t_2 from e_2 using a metric. Then they perform beta-reduction on the concrete expression $(\lambda x.\, t_1)\ t_2$ and add the resulting term to the e-graph. This is fast because it does away entirely with the need to substitute over the course of multiple rewrites. However, it only produces a single substituted term, which gives up an advantages of e-graphs: the ability to succinctly represent and act upon many identical terms. A further improvement the authors make is the utilization of deBruijn notation, which eliminates the unnecessary duplication of alpha-equivalent terms in the e-graph.

3 DECREASING THE IMPACT OF SUBSTITUTION NODES

Our chief observation is that straightforward substitution generates intermediate terms - e.g. the substitution nodes in Figure 2b - which are wholly useless. Once the substitution has been carried out, these nodes just bloat the e-graph. In short, we want beta-reducing the e-graph in Figure 2a to produce the e-graph in Figure 3b.

3.1 Reduce generated substitution nodes using a free variable check

The first and simplest optimization we can make is to perform an analysis of each e-class that tracks the free variables contained within. If we observe that x is not free in an e-class e_1, we can rewrite $e_1[x \mapsto e_2] \Rightarrow e_1$. This is because even if we carried out the entire substitution, we would not substitute anything because x is not free in e_1. In our running example, this optimization produces the e-graph in Figure 3a. It avoids generating substitution nodes for the parts of the e-graph corresponding to $y + y$.

In practice, much larger lambda calculus terms can have large portions which do not contain the desired free variable. This optimization can therefore prevent a number of substitution nodes from being generated that approximately scales with the size of terms.

3.2 Cull generated substitution nodes

The second optimization we can make is to remove substitution e-nodes after they have “served their purpose". This produces the e-graph in Figure 3b. Furthermore, it can be combined with the approach in Section 3.1 to further improve the reduction speed. This approach is pretty straightforward except that care must be taken to ensure that the rewrite rules are structured such that the substitution nodes are not generated more than once. Otherwise a node that is culled might be re-inserted into the e-graph.

3.2.1 When you can remove e-nodes. It is important for completeness to ensure that e-nodes are actually useless before we remove them from the e-graph. We consider a substitution e-node to be useless once we have applied all matching rewrites to it. Our current implementation of the culling optimization is incomplete because it removes a substitution e-node after a single rewrite: this means that if a substitution node refers to an e-class with more than one node, all but one substitution will be ignored. Like the authors in [Koehler et al. 2022] observe, this incomplete approach is in practice often sufficient; however, we suspect that making it complete will not add too much overhead.

3.2.2 Proof sketch for completeness. The idea for why it’s complete to remove a substitution node after it has substituted over its entire e-class relies upon one precondition: that no rewrite rules other than substitutions target free variables directly. If this is the case, we argue that the order you apply substitution and other rewrites does not matter. If you substitute first, then the other rewrites apply to the terms where the target variable has been substituted. If you perform the other rewrites first - so long as they do not modify the target variable - applying the substitution will replace the target variable, resulting in the same e-graph as in the first order.

3.3 Whole e-graph substitution

We can avoid generating any substitution nodes by removing the substitution node entirely from our language. An enlightening conclusion, no? As you may recall from section 1.1, we justified adding substitution nodes by claiming that beta-reduction is hard to define over an e-graph. In our experience this is correct; however, hard does not mean impossible. We were able to write a beta-reduction that produces the e-graph in Figure 3b and is more performant than the naive small-step semantics.

3.3.1 Incompleteness. We were not able to figure out a clear way to substitute in the presence of cycles: in those cases we simply did not perform a substitution. We suspect that there may be a clever way to resolve self-referential substitutions; however, even without handling these cases, the whole e-graph substitutions show promise.

4 EXPERIMENTAL FINDINGS

We tested four combinations of lambda calculi and rewrite rules on several simple benchmarks adapted from the ones in [Willsey et al. 2021].

The first two are baselines. The first baseline is a standard lambda calculus with the naive small-step rewrite rules from [Willsey et al. 2021]. The second is a deBruijn notation lambda calculus with the extraction-based substitution from [Koehler et al. 2022].

The second two are optimizations applied to the first baseline. The first optimized version adds both the culling optimization and free variable optimization from Sections 3.1 and 3.2. The second replaces the small-step rewrite rules with the whole e-graph substitution from 3.3.

We find that in benchmarks our optimized versions consistently outperform the first baseline and often outperform the second. The optimized version that incorporates culling and the free variable optimization outperforms both in all of our tests, but these results are preliminary and the tests are few in number.

We believe that both optimized versions show promise, but among the two the simplest and seemingly most effective is the combination of the culling and free variable optimizations.

5 FUTURE WORK

We seek to fix the issues of completeness in Sections 3.2 and 3.3 and investigate whether there are any meaningful trade-offs between completeness and performance.

We also intend on exploring other optimizations. One method is using alternate representations of lambda terms such as deBruijn notation or a combinator calculus such as the Routers calculus presented in [Liang et al. 2010]. This approach would prevent the issue of alpha-equivalence by choice of lambda calculus. Another is a pulsing or phase-based approach similar to the one taken in [Kourta et al. 2022]. This approach would alternate between running rewrites pertaining to substitution nodes to saturation, culling substitution nodes, and running all other rewrites in order to keep the e-graph from growing too big.

6 ACKNOWLEDGEMENTS

We want to thank Zheng Guo for helpful input on an earlier draft.

REFERENCES

Thomas Koehler, Phil Trinder, and Michel Steuwer. 2022. Sketch-Guided Equality Saturation: Scaling Equality Saturation to Complex Optimizations of Functional Programs. https://doi.org/10.48550/arXiv.2111.13040 arXiv:2111.13040 [cs].

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

Percy Liang, Michael Jordan, and Dan Klein. 2010. Learning Programs: A Hierarchical Bayesian Approach. 639–646.

Gus Henry Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Luis Ceze, and Zachary Tatlock. 2021. Pure Tensor Program Rewriting via Access Patterns (Representation Pearl). In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming* (Virtual, Canada) *(MAPS 2021)*. Association for Computing Machinery, New York, NY, USA, 21–31. https://doi.org/10.1145/3460945.3464953

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 (Jan. 2021), 23:1–23:29. https://doi.org/10.1145/3434304