Slotted E-Graphs

Rudi Schneider
r.schneider@tu-berlin.de
Technische Universität Berlin
Germany

Thomas Kœhler
thomas.koehler@inria.fr
Inria
France

Michel Steuwer
michel.steuwer@tu-berlin.de
Technische Universität Berlin
Germany

Abstract

Representing languages with bound variables in e-graphs is not straightforward. Using plain names results in reduced sharing, as multiple terms that are equivalent up to renaming are represented redundantly in the e-graph. De-Bruijn indices suffer from the same problem. Furthermore, rewriting can trigger the need to rename variables (or shift indices), such as when performing β-reduction, which can dramatically increase the size of the e-graph.

In this work, we present a novel approach to represent bound variables in e-graphs by making them a built-in feature of the data structure. In our *slotted e-graph*, e-classes are parameterized by *slots* abstracting over all free variables. Referring to an e-class now requires instantiating it by assigning a name from the users context to each slot. Renaming variables corresponds simply to different instantiations of an e-class.

Representing variables and β-reduction efficiently is an important topic in many applications of equality saturation, and we hope that this talk will spark interest with the audience of the EGRAPHS workshop.[1]

1 Introduction

Egg [5] has sparked a recent resurgence of interest into e-graphs and equality saturation, which is used for increasingly more ambitious applications. One currently open question is how to efficiently encode programs with *bound variables* in e-graphs. Glenside [4] avoids representing bound variables altogether, using a combinator-only style to represent programs. While this can work well in restricted domains, it is not a satisfying general solution. Alternatively, bound variables can be represented by name, as done in the original egg paper [5] or using de Bruijn indices [1][3].

Before introducing our novel approach, we briefly review these established options. In particular, we focus on their sharing characteristics, and how well they allow us to do fundamental operations like β and η reductions.

Named Variables. Using named variables is perhaps the most intuitive approach to representing variables. Yet, problems arise when variable names start colliding, which often makes it necessary to rename a bound variable. First, renaming a variable in an e-class requires creating an altered copy of said e-class, resulting in the duplication of potentially large parts of the e-graph and reducing sharing. Second, picking the new variable name is not an easy choice. If we pick a globally fresh name, then each new invocation of the rewrite rule will yield a different variable, constituting for unacceptable amounts of copies. On the other hand, any fixed naming scheme is prone to later collisions.

Choosing a new variable name is not only necessary for naming collisions that come up during β reductions. But additionally, every rewrite rule that creates new binders, like η expansion, will face the same conundrum.

De Bruijn Indices. Both, the problem of colliding names, and the need to generate fresh names, are solved by using De Bruijn indices. But unfortunately, as both β and η reduction will eliminate a binder, the indices of all free variables need to be shifted by one. Similarly to renaming operations, shifting operations result in duplication and reduced sharing in the e-graph.

On top of their individual characteristics, both name and index approaches lead to duplication due to the inability to merge e-classes across the boundaries of their names (or indices). The example from Figure 1 illustrates this problem on a name-based e-graph. Due to the way variables are encoded, the e-graph cannot abstract over $x - y$ and $y - x$, even though they are equivalent up to renaming. The e-graph would look the same for De Bruijn indices (e.g. replacing x with index 0, and y with index 1).

2 Slotted E-Graphs

The key idea of slotted e-graphs is to parameterize each e-class by the free variables of the terms they represent. Hence,

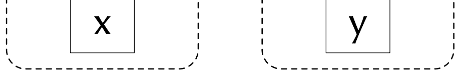

Figure 1. The term $(x - y) \cdot (y - x)$ represented in a conventional name-based e-graph.

[1] https://pldi24.sigplan.org/home/egraphs-2024#About

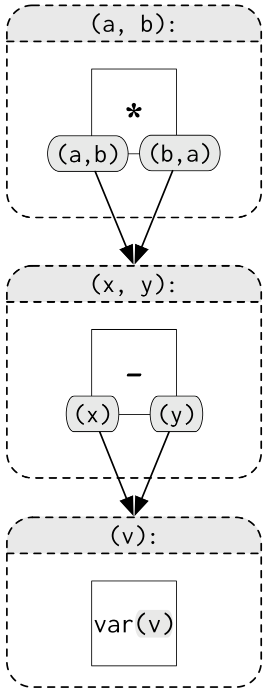

Figure 2. The term $(x - y) \cdot (y - x)$ represented in a slotted e-graph.

each free variable corresponds to an argument (dubbed *slot*) of that e-class. The central property is the following:

Definition 2.1 (Name-independent Congruence Invariant)**.** If two terms differ only in the names of (bound or free) variables, the slotted e-graph is guaranteed to represent them using the same e-class.

To visualize the idea, we revisit the term $(x - y) \cdot (y - x)$ from Figure 1, and compare it to its slotted equivalent in Figure 2.

The e-class at the bottom represents single-variable terms like x and y, abstracting over the concrete variable through the slot v. The special e-node $var(v)$ is used to signify that this e-class represents the variable indicated by the slot v.

In the middle, we have the e-class representing terms like $x-y$. As this term has two free variables, this e-class requires two slots x and y. It contains a single subtraction e-node, which invokes the single-variable e-class once with x and once with y, representing $x - y$.

Finally, the upper e-class represents the whole term $(x - y) \cdot (y - x)$. As we again have two free variables, we require two slots. In this case, we call them a and b to avoid confusions with the slots of the previous e-class. Whether a slot a in this e-class corresponds to x or y in the next e-class is solely decided by the way the e-class is invoked. This final e-class consists of a multiplication e-node that invokes our subtraction e-class once with (a, b) and once with (b, a) to obtain $a - b$ and $b - a$ respectively.

It is important to stress, that the names we choose for our slots are only intended to aid readability. The names are arbitrarily chosen and local to the e-class.

2.1 Bindings

So far, we only worked with free variables, without even mentioning how to *bind* them. A typical example for a binder is the λ-abstraction. We use it to examplify how binders work in a slotted e-graph.

To visualize how λ-abstraction works, we again consider an example.

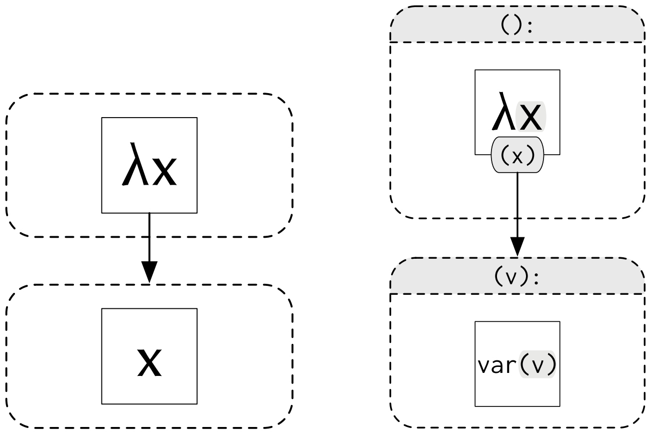

Figure 3. The identity function $\lambda x.x$ in both a conventional e-graph, and a slotted e-graph.

In the conventional name-based e-graph, the mapping between the variable and its binder happens *per name*. Instead, with slotted e-graphs this connection is expressed using slots. A binder e-node λx declares a new slot x, that can be used to invoke other e-classes. The key advantage is that each binder e-node can directly invoke any e-class with its slot, independent of the internal slot names. This is a stark contrast to named variables, where a binder can only refer to a variable if their names agree.

A very similar method of expressing the connection between a binder and its variable is discussed in [2].

2.2 Conflicting sets of free variables

A notable difficulty for slotted e-graphs arises from the fact that equal terms might not necessarily agree on a set of free variables. This is critical, as our e-classes are intended to abstract over these equivalent terms, exposing their free variables as slots. Consider an e-class containing both $y \cdot 0$ and 0. What set of free variables should it expose?

Our answer is that this e-class should not expose any free variables, because the variable y does not contribute to the result of the expression. Generally, if two terms are *equal*, while one of them does not depend on some variable, then clearly the other term does neither. Formally, if $f(x, y) =$

$g(x)$ for all x, y, then we know that f can not depend on y, as $f(x, y) = g(x) = f(x, c)$ for any constant c. Hence, the set of free variables exposed by an e-class is the intersection of the free variables exposed by its terms.

However, this still leaves the question unanswered: What happens to y when we unify $y \cdot 0$ with 0. We will again, show an illustration.

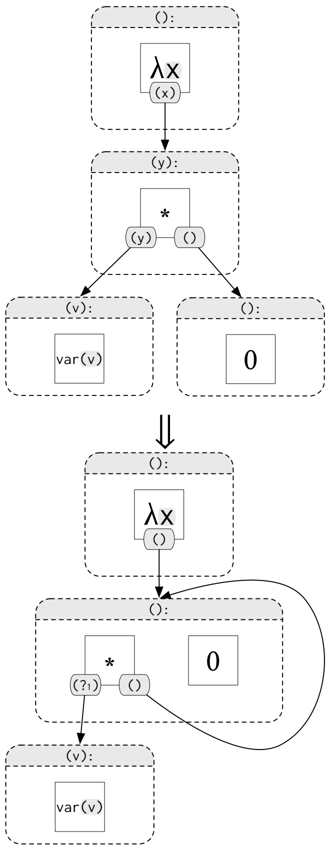

Figure 4. The term $\lambda y.y \cdot 0$ in a slotted e-graph, before and after unifying $y \cdot 0 = 0$.

After unifying $y \cdot 0 = 0$, the slot y became the *redundant slot* ?1. Redundant slots represent variables that have no impact on the resulting expression, but are still referenced in some (but not all!) of its terms. We do not just replace redundant slots by any fixed constant c as suggested before, because we still want future lookups of $v \cdot 0$ for any variable v to match this e-node.

3 The slotted e-graph data structure

Due to our ambitious extension of e-graphs to encode binders and variables as first-class citizens in the language, we need to extend the conventional e-graph data structure at a few key spots.

3.1 Name-independent E-Node Lookup

A conventional e-graph has the key sharing guarantee that each e-node is contained in at most one e-class. This can be ensured by the hashcons, a map that hashes an e-node and stores in which e-class it is contained.

However, in the slotted e-graph, e-nodes do not simply refer to ids, but ids equipped with a list of slot names. As these slot names are arbitrary and do not matter, we need to eliminate them before hashing. Otherwise, two conceptually identical e-nodes with different slot names would not be considered equivalent by our hashcons mechanism. We eliminate the slot names, by computing a naming normal-form of our e-node, called its *shape*. The shape is obtained by enumerating all of its slot names based on the order of their first occurence in the e-node, and renaming them accordingly to $s1, s2, \ldots$.

Note that this slot-normalization should not be confused with the Id-normalization based on the unionfind data structure!

3.2 Unification of E-Classes with slots

A union between two e-classes a and b can typically be understood as adding the equation $a = b$ to your knowledge base. However, in a slotted e-graph, this becomes more nuanced, as there are multiple ways to equate two ids, based on how you connect their slots. So, a union between a and b would correspond to an equation like $a(x, y, z) = b(y, x)$ where x, y and z are slot names. This causes multiple challenges.

Extended unionfind. First of all, a simple unionfind data structure is not enough anymore. In addition to the fact that we normalize a to b, we also need to store how their slots need to be permuted. Thus, the original unionfind of type `Id -> Id` becomes a mapping of type `Id -> (Id, SlotMap)`.

Redundant Slots. The reader might recall the $x \cdot 0 = 0$ example: In cases where both sides of the equation have different sets of slots, we introduce redundant slots for their symmetric difference.

Self-Symmetries. This leaves us with one final challenge. What happens with unions of the form $a(x, y) = a(y, x)$, i.e. *self symmetries*. Typically, when encountering an equation $a(\ldots) = b(\ldots)$, we replace all occurences of a with b, or vice versa. This is not possible with self-symmetries.

There are multiple approaches to tackle this problem, but the most promising one seems to store a permutation group for each e-class. In that case, a union like $a(x, y) = a(y, x)$ would simply add a permutation to that group. However, it is worth noting that this makes computing the shape (i.e. slot normal form) of an e-node harder, due to a new degree of freedom.

References

[1] Nicolaas Govert De Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes mathematicae (proceedings)*, Vol. 75. Elsevier, 381–392.

[2] Dan R Ghica. 2021. Operational semantics with hierarchical abstract syntax graphs. *arXiv preprint arXiv:2102.02363* (2021).

[3] Thomas Koehler, Andrés Goens, Siddharth Bhat, Tobias Grosser, Phil Trinder, and Michel Steuwer. 2024. Guided Equality Saturation. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 1727–1758.

[4] Gus Henry Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael B. Taylor, Luis Ceze, and Zachary Tatlock. 2021. Pure tensor program rewriting via access patterns (representation pearl). In *MAPS@PLDI 2021: Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming, Virtual Event, Canada, 21 June, 2021*, Roopsha Samanta and Isil Dillig (Eds.). ACM, 21–31. https://doi.org/10.1145/3460945.3464953

[5] 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.