Slotted E-Graphs

First-Class Support for (Bound) Variables in E-Graphs

RUDI SCHNEIDER, Technische Universität Berlin, Germany
MARCUS ROSSEL, Barkhausen Institut, Germany
AMIR SHAIKHHA, University of Edinburgh, United Kingdom
ANDRÉS GOENS, University of Amsterdam, Netherlands
THOMAS KŒHLER, ICube Lab, CNRS, Université de Strasbourg, France
MICHEL STEUWER, Technische Universität Berlin, Germany

Equality saturation has gained significant interest as a powerful optimization and reasoning technique. At its heart is the e-graph data structure, that space-efficiently represents equal sub-terms uniquely. An important open problem in this context is extending this efficient representation to languages featuring (bound) variables. Independent of how we represent variables in e-graphs, either as names or nameless (using de Bruijn indices), sharing is broken as sub-terms that differ only in the names of their variables are represented separately. This results in aggressive e-graph growth, bad performance, as well as reduced expressiveness.

In this paper, we present a novel approach to representing bound variables in e-graphs by making them a first-class built-in feature of the data structure. Our *slotted e-graph* represents terms that differ only by (bound or free) variable names uniquely. To do so, e-classes that represent equivalent terms via e-nodes are parameterized by *slots*, abstracting over free variables of the represented terms. Referring to an e-class from an e-node now requires relating the variables from its context to the slots of the e-class.

Our evaluation of slotted e-graph uses two case studies from compiler optimization and theorem proving to show that performing equality saturation for languages with bound variables is greatly simplified and that we can solve practically relevant problems that cannot be solved with e-graphs using de Bruijn indices.

CCS Concepts: • **Theory of computation → Equational logic and rewriting**.

Additional Key Words and Phrases: binders, e-graphs, equality saturation, rewrite systems

ACM Reference Format:
Rudi Schneider, Marcus Rossel, Amir Shaikhha, Andrés Goens, Thomas Kœhler, and Michel Steuwer. 2025. Slotted E-Graphs: First-Class Support for (Bound) Variables in E-Graphs. *Proc. ACM Program. Lang.* 9, PLDI, Article 223 (June 2025), 23 pages. https://doi.org/10.1145/3729326

1 Introduction

Equality Saturation [Tate et al. 2009] is a technique that enables exploring many different ways to rewrite terms efficiently, thanks to the clever *e-graph* (equivalence graph) data-structure that represents many equivalent terms compactly [Nelson and Oppen 1980]. The strength of equality saturation is that it relieves users from having to decide in which order rewrite rules should be applied, which is often difficult.

Authors' Contact Information: Rudi Schneider, Technische Universität Berlin, Berlin, Germany, r.schneider@tu-berlin.de; Marcus Rossel, Barkhausen Institut, Dresden, Germany, marcus.rossel@barkhauseninstitut.org; Amir Shaikhha, University of Edinburgh, Edinburgh, United Kingdom, amir.shaikhha@ed.ac.uk; Andrés Goens, University of Amsterdam, Amsterdam, Netherlands, a.goens@uva.nl; Thomas Kœhler, ICube Lab, CNRS, Université de Strasbourg, Strasbourg, France, thomas.koehler@cnrs.fr; Michel Steuwer, Technische Universität Berlin, Berlin, Germany, michel.steuwer@tu-berlin.de.

ACM 2475-1421/2025/6-ART223
https://doi.org/10.1145/3729326

The egg library by Willsey et al. [2021] has sparked a renewed interest in equality saturation as an optimization and reasoning technique. There is a growing community expanding the foundation of the technique and exploring practical applications in various domains, ranging from optimizing software [Kœhler et al. 2024; Shaikhha et al. 2024; Smith et al. 2021] and hardware [Cheng et al. 2024; Wang et al. 2023], to reasoning about correctness in theorem proving [Kœhler et al. 2024; Rossel 2024] and verification [Dickerson et al. 2024].

To use equality saturation, the initial terms, written in the language of the application domain, have to be represented in the e-graph. One central challenge is how to represent variables. With variables, we mean *names that have different meanings in different terms depending on their context.* In contrast, we call names that always have the same meaning independent of their context constants, such as `PI` when used to always name the circle constant π. Variables are present in almost all languages of interest, and are often introduced by syntactic constructs that *bind* a name in a particular scope, such as a lambda-binding: $\lambda x.\ t$ where the variable x is bound in the term t. Let us see how different projects approach the representation of variables in e-graphs.

The Glenside project by Smith et al. [2021] avoids the issue of variables altogether with a combinator-only language design without variables, as *"name bindings significantly complicate term rewriting"* and the authors state that they *"have found the additional complexity and rewrite search space blow up substantially eliminate the potential advantages of term rewriting in such IR designs"*. Of course, a combinator-only language without variables has downsides in itself. Besides being unfamiliar to users who are used to variables, it is often impractical to translate terms with variables into combinator-only style, as in the worst case it results in a term size of $O(n^3)$ [Lachowski 2018].

In the egg paper, an encoding of lambda calculus is presented that represents variables with their names encoded as plain strings [Willsey et al. 2021]. This representation is presented as *"contrived"* and name bindings are described as *"a perennially tough problem for e-graphs"*. The name-based representation has a couple of obvious downsides, as two equal terms that only differ by their bound variable names are stored twice and not treated as equal. Furthermore, problems arise when variable names collide during rewriting, e.g., when performing capture-avoiding substitution, making it necessary to rename variables and duplicate large parts of the e-graph. Detecting these collisions during rewriting is not trivial, requiring delicate rules and tracking which variables are free in a context via a dedicated analysis. Besides these complexity challenges, due to the frequent renaming the e-graph can rapidly grow, quickly running out of memory as shown by Kœhler et al. [2024].

In their paper, Kœhler et al. [2024] use *de Bruijn indices* [De Bruijn 1972] to represent bound variables in e-graphs. De Bruijn indices encode variables as numbers counting the numbers of binders in scope between the occurrence of the variable and its corresponding binder. This encoding of variables solves the problems that occur when representing variables by their names, but unfortunately, introduces new problems. When rewrites eliminate or introduce binders, all indices have to be shifted, using a dedicated set of rewrites, to maintain correctness. Getting shifting right is a tedious task and similar to renaming, shifting results in unnecessary duplication that can blow up the e-graph size quickly, as we will discuss in more depth in Section 4.

Representing bound variables in e-graphs without these complexities and scaling problems is considered an important research question, and the egg paper acknowledges, that *"better support for languages with binding is important future work"* [Willsey et al. 2021].

In this paper, we address this outstanding open research question by making bound variables a first-class built-in feature of the e-graph data-structure. We introduce *slotted e-graphs*, which represent terms that differ only by (bound or free) variable names uniquely. To do so, e-classes, which group equivalent terms, are parameterized by *slots* abstracting over all free variables of the equivalent terms represented by its e-nodes. Referring to an e-class from an e-node now requires relating the variables from the e-node's context to the slots of the e-class.

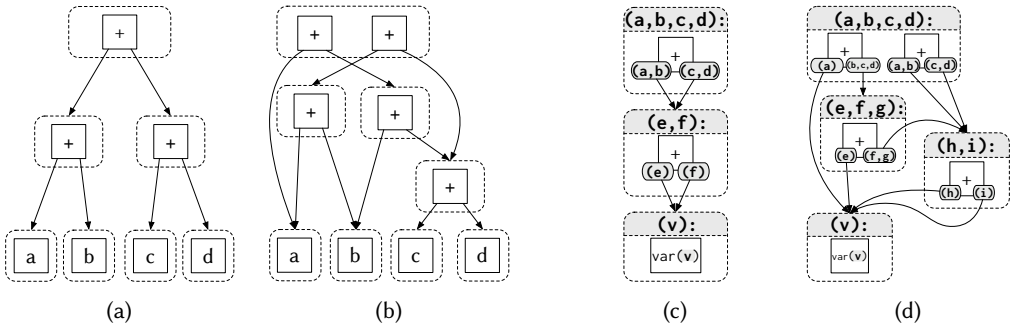

Fig. 1. The term $(a + b) + (c + d)$ over variables a, b, c, and d, is represented in a conventional e-graph in (a) and the *slotted e-graph* in (c). (b) and (d) show the e-graphs after the equivalent term $a + (b + (c + d))$ has been added. The subterm $c + d$ is shared and stored only once in both conventional e-graphs, but $a + b$ is stored separately. In the slotted e-graph all subterms of the form $x + y$ (where x and y are variables) are stored only once, as slotted e-graphs guarantee that terms which differ only by variable names are represented in the same unique e-class.

We formalize slotted e-graphs and the extended congruence modulo renaming relation it maintains. For clarity, we will focus throughout the paper on examples using lambda calculus as a familiar language foundation, but slotted e-graphs are capable of representing languages with arbitrary binders, not just lambda calculus.

We have implemented slotted e-graphs in an open-source library called `slotted`[1] that provides convenient support for performing equality saturation in languages with variables, as variables are directly represented as slots and don't need to be specially encoded, e.g., using de Bruijn indices, eliminating the need for extra rules for shifting. Writing rewrite rules is greatly simplified, as rules can be directly expressed using familiar notation without having to worry about name collisions, no e-class analysis is required for keeping track of bound and free variables, and built-in syntax is provided for performing substitutions.

To summarize, our contributions are:

- We introduce *slotted e-graphs*, that add first-class support for variables to e-graphs (Section 2);
- we formalize slotted e-graphs and the extended congruence modulo renaming relation it maintains (Section 3);
- we evaluate our implementation of slotted e-graphs, called `slotted`, by systematically comparing its capabilities to conventional e-graphs, and then by discussing two practical case studies from theorem proving and compilation (Section 4).

2 Slotted E-Graphs, Informally

An e-graph (equivalence graph) is a data structure that compactly represents equivalence classes of terms in a given language. This is achieved by storing equivalent subterms only once, combined with a congruence relation that establishes a notion of equivalence (beyond pure syntactic equality).

Figure 1a shows an e-graph representing the term $(a + b) + (c + d)$, with a, b, c, and d denoting variables. When we add the equivalent term $a + (b + (c + d))$ to the e-graph in Figure 1b, we now represent two syntactically different, but equivalent terms. This is represented by placing the two terms into the same *e-class* (equivalence class), the dotted box at the top. An e-class contains a

[1]See here: https://crates.io/crates/slotted-egraphs

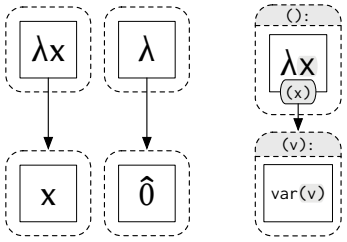

Fig. 2. The identity lambda in a conventional, and slotted e-graph where the slot x is *internal* to the λ.

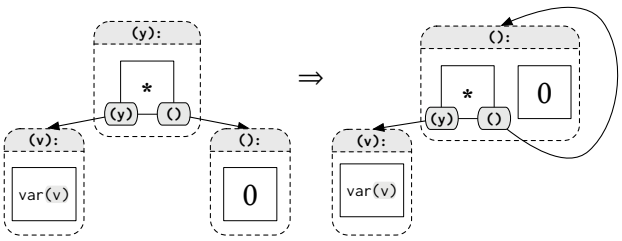

Fig. 3. The term $y \cdot 0$ in a slotted e-graph, before and after unifying $y \cdot 0 = 0$. Slot y becomes *redundant* and does not appear in the (empty) parameter list of the merged e-class.

set of equivalent *e-nodes*, each representing one equivalent term (or set of terms, more generally). The children of e-nodes are e-classes, reflecting the notion that we can pick any e-node within an e-class, as they all represent equivalent terms.

The e-graph in Figure 1b represents the two equivalent terms $(a+b)+(c+d)$ and $a+(b+(c+d))$. We can easily observe the compact, space-efficient sharing for the subterm $c+d$, which is indeed only stored once. However, if we look closely at Figure 1b, we can see that $a+b$ and $c+d$ are stored separately. We argue that this representation is missing out on further sharing opportunities, as these terms are exactly the same modulo variable names.

The key insight and idea that motivates our modified *slotted e-graph* data structure is the intuition that *names of variables should not matter*: $(a+b)+(c+d)$ and $(w+x)+(y+z)$ are exactly the same terms if we just rename the variables.

Figure 1c shows the term $(a+b)+(c+d)$ in our slotted e-graph, with Figure 1d again adding the equivalent term $a+(b+(c+d))$. In a slotted e-graph, e-classes are parameterized by the free variables of the terms they represent. Each free variable corresponds to a parameter, we call *slot*, of that e-class. The terms $a+b$ and $c+d$ are now represented by a single parameterized e-class. In fact, the central property of a slotted e-graph is that *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-node.* E-nodes in slotted e-graphs must now explain how the slots of their surrounding e-class relate to the parameters of the referenced e-classes. We visualize the slots of e-classes like parameters of a function, and when referring to an e-class from an e-node we use a notation similar to a function invocation, instantiating these parameters.

2.1 Bindings

So far, we only discussed free variables, without mentioning how to *bind* them. The most typical example for a binder is the λ-abstraction. We use it to exemplify how binders work in a slotted e-graph. In a conventional e-graph, the mapping between the variable and its binder happens per name or de Bruijn index, depending on the encoding, as shown in Figure 2 on the left. In a slotted e-graph, the binding connection is expressed using slots, as shown on the right of Figure 2. A binder e-node $\lambda\, x$ introduces a new slot x, that is then used to instantiate other e-classes. The key difference is that the name of the slot is internal and cannot be referenced by other e-nodes – the surrounding e-class does not expose the slot and has the empty parameter list `()`. Thus, the slotted e-graph represents *all* identity lambdas, whereas using the named representation in the conventional e-graph does not consider $\lambda x.\ x$ and $\lambda y.\ y$ as equal. The de Bruijn encoding in a conventional e-graph also represents all identity lambdas as $\lambda.\ \hat{0}$. However, as Maziarz et al. [2021] pointed out, this is not true in general as de Bruijn indices do not guarantee that α-equivalent

subterms are represented equally. For example, the term $\lambda t.\ foo\ (\lambda x.\ x + t)\ (\lambda y.\ \lambda x.\ x + t)$ clearly contains the expression $\lambda x.\ x + t$ twice, but in the de Bruijn encoding they are encoded differently: $\lambda.\ foo\ (\lambda.\ \hat{0} + \hat{1})\ (\lambda.\ \lambda.\ \hat{0} + \hat{2})$.

2.2 E-Classes Representing Terms with Different Numbers of Free Variables

An interesting question arises for slotted e-graphs from the fact that equal terms might have different sets of free variables. This is critical, as our e-classes are intended to abstract over these equivalent terms, exposing their free variables as slots. Figure 3 shows on the left a slotted e-graph for the term $y \cdot 0$. When we expose the fact that $y \cdot 0$ is equal to 0 to the slotted e-graph, we must merge the two e-classes representing these subterms. However, which set of free variables should the new e-class 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 also does not actually depend on this concrete variable. We justify our choice because, if $\forall x, y.\ f(x, y) = g(x)$, then we know that f cannot depend on y, as $f(x, y) = g(x) = f(x, z)$ for any other variable z. Hence, we expose the intersection of the free variables of its terms as the set of free variables of the e-class.

Figure 3 shows on the right the slotted e-graph resulting from merging $y \cdot 0$ with 0: the slot y has become a *redundant slot*. Redundant slots represent variables that have no impact on the resulting expression, but are still referenced in some (but not all) of its terms. When we look up terms in the slotted e-graph, redundant slots can be matched for any variable, so that the graph from Figure 3 represents the terms: 0, $y \cdot 0$, but also $x \cdot 0$, and generally all multiplications of *any* variable with 0.

2.3 Representing Symmetries

Another challenge of slotted e-graphs is how to represent symmetries of variables. If we have the term $a + b$, we might want to apply commutativity to record that $a + b = b + a$. In a conventional e-graph, we would now merge the e-class containing $a + b$ and the e-class containing $b + a$, resulting in an e-class with two addition e-nodes. However, in a slotted e-graph, both terms are already represented by the same class, simply invoked with a different order of arguments. To represent this in slotted e-graphs, each e-class stores a permutation group that records all allowed symmetries of its slots. Figure 4 shows the slotted e-graph, where the e-class can now be invoked either with (a, b) or (b, a), representing the same equivalence class of terms.

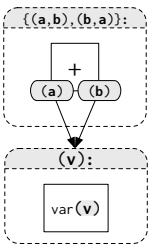

Fig. 4. The terms $a + b$ and $b + a$ represented in a slotted e-graph.

Representing symmetries using permutation groups instead of simply storing multiple permuted copies of the e-nodes directly has several advantages:

(a) Permutation groups are efficient. In our implementation, we use the Schreier-Sims algorithm [Seress 2003], which allows us to represent exponentially many permutations using a polynomial amount of space. This scales better than storing a copy for each e-node *times* each permutation from your group directly.

(b) Symmetries affect the equivalence relation of the slotted e-graph. Conventional e-graphs evaluate whether two terms are equivalent by lowering them to e-classes (or practically, e-class ids), and evaluating whether these e-classes are equivalent. Slotted e-graphs lower terms to *invocations* of e-classes instead. Hence, we need an equivalence relation between invocations of e-classes and, therefore, in order to answer whether two different invocations of the same e-class are equivalent or not, we are required to factor in the symmetries of that e-class.

3 Slotted E-Graphs

In this section, we make the intuitions more concrete, presenting a formal definition of the e-graph data structure extended with slots, and explaining how algorithms for manipulating it are affected.

3.1 Slotted Terms and the Congruence Modulo Renaming Relation

We first discuss the base language of terms that will be represented in slotted e-graphs. E-graphs consider ground terms in a term language. With slotted e-graphs, we want to also represent (free and bound) variables in our terms. We formulate the syntax of the term language using symbolic expressions, like the formalization of egg [Willsey et al. 2021], with the addition of representing free and bound variables in our terms.

function symbols f, g

slots $\$x, \$y, \ldots$

term children $tc ::= t \mid tc_1, \ldots, tc_k \mid \text{bind } \$x\ tc \mid \$x \qquad k \geq 1$

terms $t ::= f \mid f(tc)$

Fig. 5. The syntax of slotted terms. Extensions of ground terms in e-graphs are marked in green.

Figure 5 shows the basic syntactic constructs for building slotted terms. Extensions over traditional terms are marked with a green box. These extensions are the slots, which are variables prefixed by a \$, and a special constructor, bind $\$x\ tc$, that denotes that the variable $\$x$ is bound in the (sub-)term(s) in tc.

Slotted terms correspond very closely to *nominal terms* in the context of nominal rewriting, which is a family of techniques for dealing with terms modulo α-equivalence. The core idea of nominal rewriting is using permutations instead of substitution, and reasoning about freshness constraints explicitly within the nominal reasoning. This avoids many of the pitfalls of capturing in substitution. Based on this approach, congruence and unification modulo α equivalence, are defined with algorithms that deal with explicit namings (and renamings) instead of de Bruijn indices [Fernández and Gabbay 2007]. Our notion of these concepts are very similar to, and inspired by these.

Example 1 (The Language of Lambda Calculus)**.** Lambda calculus contains four types of terms: λ-abstraction, function application, variables and let. In the following, lambda, app, var and let are the *function symbols* from Figure 5.

$\lambda\ \$x.\ t_1$	is encoded as	$\text{lambda}(\text{bind}\ \$x\ t_1)$
$t_1\ t_2$	is encoded as	$\text{app}(t_1, t_2)$
$\$x$	is encoded as	$\text{var}(\$x)$
$\text{let}\ \$x\ = t_1\ \text{in}\ t_2$	is encoded as	$\text{let}(t_1, \text{bind}\ \$x\ t_2)$

Note that the $\text{let}(t_1, \text{bind}\ \$x\ t_2)$ constructor has a single binder slot $\$x$, which binds every occurrence of $\$x$ in the body t_2 of the "let", but **not** in the term t_1 we substitute with.

Congruence Modulo Renaming. In this section, we clarify which terms are unified by slotted e-graphs. A conventional e-graph carries out congruence closure, so it maintains a **congruence** relation. A slotted e-graph maintains a more general relation that takes variables and bindings into account: it maintains a **congruence modulo renaming** relation. This is very similar to *nominal equational logic* [Urban et al. 2004]. It deals with *renamings* of *slots*, which we define as follows:

Definition 1 (Slots of a Term)**.** The slots of a term t are defined as follows:

$$\text{slots}(tc_1, \ldots, tc_k) := \bigcup_{i=1}^{k} \text{slots}(tc_i) \qquad \text{slots}(\$x) := \{\$x\} \qquad \text{slots}(f) := \emptyset$$
$$\text{slots}(\text{bind } \$x\ tc) := \text{slots}(tc) \setminus \{\$x\} \qquad \text{slots}(f(tc)) := \text{slots}(tc)$$

If we interpret slots as variables in terms, then the set of slots of a term is just the set of free variables that appear in that term.

Definition 2 (Slot Renamings)**.** We consider *renamings* m, which are bijections from one set of slots to another. We write these renamings as a list $m := [\$x_1 \mapsto \$y_1, \ldots \$x_n \mapsto \$y_n]$. We define the domain $\text{dom}(m) := \{\$x_1, \ldots, \$x_n\}$ and image $\text{im}(m) := \{\$y_1, \ldots, \$y_n\}$ accordingly.

$$m * (tc_1, \ldots, tc_k) := (m * tc_1, \ldots, m * tc_k) \qquad m * \$x := m(\$x) \qquad m * f(tc) := f(m * tc)$$
$$m * (\text{bind } \$x\ tc) := \text{bind } \$f\ (m * (\$x\ \$f) * tc), \qquad \text{where } \$f \text{ fresh}$$

Here, $(\$x\ \$f)$ is the renaming mapping $\$x$ to $\$f$ and vice-versa, and is the identity otherwise (it is a transposition in cycle notation, cf. [Lang 2012]). Note that $*$ is also defined for renamings where the domain is not equal to the image.[2] Further, the operation $m * x$ is only defined if $\text{slots}(x) \subseteq \text{dom}(m)$ to ensure that every slot in x has an associated slot in m. In our implementation, renamings are represented by a list of key-value pairs sorted by the lexicographical order of their key slots.

Congruence Modulo Renaming. An e-graph takes a set of unions (i.e. term equations) to answer equality queries quickly. We denote with E the set of given term equations of the form $a \approx b$. In an e-graph, these are all *ground* equations, in slotted e-graphs, they can also include variables.

The congruence modulo renaming relation builds on the regular congruence of terms (Figure 6a) that combines the standard definition of an equivalence relation (symmetric, reflexive and transitive) with congruence of function symbols. We adapt this relation to also cover renaming (Figure 6b).[3] Further, we add the necessary rules to extend congruence to our other syntactic constructs, as well as α-conversion and (slot) renaming. The latter two rules are covered by a single rule that renames both bound or free variables in nominal rewriting [Urban et al. 2004].

Justifying Redundancy. We revisit the example from Figure 3 to derive *redundancy* from the rules of the congruence modulo renaming relation. Consider $E = \{\text{mul}(\$x, 0) \approx 0\}$. We can prove $E \vdash \text{mul}(\$y, 0) \approx 0$ by applying the "renaming" rule with $[\$x \mapsto \$y]$ on $\text{mul}(\$x, 0) \approx 0$, obtaining

$$E \vdash \text{mul}(\$x,\ 0)\ \approx\ 0\ \approx\ \text{mul}(\$y,\ 0).$$

In other words, $mul(\$x,\ 0)$ is equivalent to 0 but also to $mul(\$y,\ 0)$, $mul(\$z,\ 0)$ etc. Hence, we call $\$x$ *redundant*. Notably, this works even if $\$x$ and $\$y$ are bound. We can prove:

$$E \vdash \lambda\$x.\, \text{mul}(\$x,\ 0)\ \approx\ \lambda\$x.\, \text{mul}(\$y,\ 0),$$

using the previous proof and the "cong. (bind)" rule. Notably, here the bound $\$x$ becomes a free $\$y$, but it is still sound because of the equality with the term 0 where both variables do not appear. More generally, once we have found an equation that establishes a slot is *redundant*, we can replace it with any other slot (bijectively) in all contexts. Hence the original slot has lost its meaning. For this reason, we do not keep redundant slots in our e-classes.

[2] More precisely, this defines a partial action [Kellendonk and Lawson 2004] of the inverse semigroup of partial permutations on the set of variables (cf. [Lawson 1998]), but induces also a left action of the symmetric group considered as a subsemigroup.
[3] as mentioned above, very closely related to nominal equational logic [Urban et al. 2004]

$$\text{start}\ \frac{a \approx b \in E}{E \vdash a \approx b} \qquad \text{reflexivity}\ \frac{}{E \vdash a \approx a} \qquad \text{symmetry}\ \frac{E \vdash a \approx b}{E \vdash b \approx a}$$

$$\text{transitivity}\ \frac{E \vdash a \approx b \quad E \vdash b \approx c}{E \vdash a \approx c} \qquad \text{congruence}\ \frac{E \vdash a_i \approx b_i,\ i = 1, \ldots, n}{E \vdash f(a_1 \ldots a_n) \approx f(b_1 \ldots b_n)}$$

(a) Congruence Relation

$$\text{cong. (variadic)}\ \frac{E \vdash tc_i \approx tc'_i,\ i = 1, \ldots, k}{E \vdash tc_1, \ldots, tc_k \approx tc'_1, \ldots, tc'_k} \qquad \text{cong. (bind)}\ \frac{E \vdash tc \approx tc'}{E \vdash \text{bind}\ \$x\ tc \approx \text{bind}\ \$x\ tc'}$$

$$\text{cong. (f)}\ \frac{E \vdash tc \approx tc'}{E \vdash f(tc) \approx f(tc')} \qquad \alpha\text{-conversion}\ \frac{\$y \text{ does not occur in } tc}{E \vdash \text{bind}\ \$x\ tc \approx \text{bind}\ \$y\ (\$x\ \$y) * tc}$$

$$\text{slot-renaming}\ \frac{E \vdash t \approx t' \qquad \text{slots}(t), \text{slots}(t') \subseteq \text{dom}(m)}{E \vdash m * t \approx m * t'}$$

(b) Congruence Modulo Renaming Relation

Fig. 6. Congruence Relation and its Slotted Extensions

slotted e-class ids a, b

e-node children $nc ::= m * a \mid nc_1, \ldots, nc_k \mid \text{bind}\ \$x\ nc \mid \$x \qquad k \geq 1$

e-nodes $n ::= f \mid f(nc)$

Fig. 7. Syntax and metavariables of Slotted E-Nodes

3.2 The Slotted E-Graphs Data Structure

In this subsection, we formally introduce what a slotted e-graph is, and how it works internally. The presentation follows closely [Willsey et al. 2021]. We mark the new additions in green.

Figure 7 describes the syntax of slotted e-nodes. The definition of e-nodes is equivalent to terms, except that we do not recursively contain subterms, but in their place we contain *renamed ids* $m * a$, where m is a renaming and a an *id*. A *renamed id* corresponds to the intuition of slot-annotated arrows going from e-node to e-class in Figures 1-4. Thus, we can think of it as an "invocation" of an e-class. The renaming m associates the slots from the caller-context to the slots of the e-class that gets called. We occasionally write $a[\$x, \$y, \ldots]$, as syntactic sugar for $m * a$ if m maps $\text{slots}(a)$ (lexicographically ordered) to $\$x$, $\$y$, ...

A key idea behind e-graphs is how they replace the recursive children in terms with ids that serve as "pointers", which can now point at e-classes instead of concrete subterms. In slotted e-graphs, where terms have slots, ids are not enough as pointers, we also need a renaming. This means that we need slotted versions of the data structures that also keep track of these renamings. Similarly, comparing e-nodes becomes more complicated because of the renamings. To deal with this, we define a renaming-invariant normal form for an e-node, called its "shape". It allows us to decompose each e-node into its canonicalized shape and a renaming. Next we introduce slotted variants of the e-graph data structures, union-find and hashcons, and explain how they interact with each other.

Definition 3 (Slotted E-Graph)**.** A slotted e-graph is a triple (U, M, H) where

- U is a slotted union-find, which provides a function find from renamed ids to equivalent renamed ids with a canonical id.
- M is a mapping from slotted e-class ids to slotted e-classes.
- H is the hashcons, a mapping from shapes to slotted e-class ids.

Definition 4 (Slotted E-Class)**.** A slotted e-class is a triple (S, B, G), where

- S is a set of slots. They represent the "free variables", or "free slots" of this e-class. The set S is drawn in the header of an e-class in Section 2.
- B is a hash map from shapes to renamings. B represents the set of e-nodes that is contained in this e-class, but decomposed into shapes. B maps the shapes contained in the e-class to a renaming, associating the slots of the shape to the slots S of this e-class. To aid intuition, we say that the e-class (S, B, G) *contains* the e-nodes $m * n$ for $(n, m) \in B$.
  Further, for all $(n, m) \in B$, we require $\text{slots}(n) = \text{dom}(m)$ and $S \subseteq \text{im}(m)$. All slots that are contained in e-nodes of this e-class (or equivalently in $\text{im}(m)$), but not in S are called *redundant*.
- G is a permutation group. G is a set of renamings with domain and image S. G expresses which symmetries this e-class has, as shown in Figure 4.

For a slotted e-class $C = (S, B, G)$, we write $C.S$, $C.B$ and $C.G$ to denote S, B and G.

We require that, for a slotted e-graph, $H[n] = a$ iff $n \in \text{dom}(M[a].B)$. This is our version of the *hashcons invariant* from [Willsey et al. 2021], ensuring that our hashcons is kept up-to-date about which e-classes contain which shape. Further, we require that our hashcons H only references leader ids (cf. Definition 5).

We can extend the definitions of the slots function and the (partial) action $m * X$ from terms and term-children to e-nodes and e-node children analogously. For the additional case of renamed ids $m * a$, we define:

$$\text{slots}(a) := M[a].S \qquad \text{slots}(m * a) := m\,\text{slots}(a) = \{m(\$y) \mid \$y \in \text{slots}(a)\}$$
$$m * (m' * a) := (m * m') * a \qquad m * m' := m \circ m'$$

Definition 5 (Slotted Union-Find)**.** The slotted union-find U is a hash map from a slotted e-class id to a renamed id. We call an id a a *leader* if there exists an m such that $U[a] = m * a$, otherwise a *follower*. Further, we require that $m = \text{id}_{M[a].S}$ holds for all leaders. We say that a renamed id $m * a$ is a leader, if a is a *leader* and $\text{dom}(m) = \text{slots}(a)$, otherwise a *follower*.

The find function converts any renamed id to an equivalent leader, computed as the fixed point:

$$\text{find}(m * a) := \begin{cases} m * a, & \text{if } m * a \text{ is a leader} \\ \text{find}(m * U[a]), & \text{if } m * a \text{ is a follower} \end{cases}$$

We use the *collapsing rule* (or "path compression") from [Tarjan 1975] to efficiently compute find.

We require $\text{slots}(U[a]) \subseteq \text{slots}(a)$ for all a. Thus, leaders have a minimal set of slots, when compared to their followers. Or conversely, leaders have a maximal set of redundant slots.

Note that while every id has a unique id as its leader, not every renamed id has a unique *renamed* id as leader. This is due to the fact that leader renamed ids might have other leader renamed ids that are equivalent based on a group symmetry. To respect this, we need to take extra care to define shapes, our canonicalized e-nodes, later on.

Example 2 (Union-Find Merge)**.** Consider two e-class ids c_1 and c_2 with $\text{slots}(c_1) = \{\$x, \$y\}$ and $\text{slots}(c_2) = \{\$a, \$b\}$. Let's assume we want to establish that $c_1[\$l, \$r]$ is equivalent to $c_2[\$l, \$r]$. This requires us to merge the underlying e-classes (for example c_2 into c_1). When merging these two e-classes, we need to establish which slot from c_1 corresponds to which slot from c_2. As we want to unify $c_1[\$l, \$r] = [\$x \mapsto \$l, \$y \mapsto \$r] * c_1$ with $c_2[\$l, \$r] = [\$a \mapsto \$l, \$b \mapsto \$r] * c_2$, we end up with the renaming $[\$a \mapsto \$x, \$b \mapsto \$y]$, by combining both renamings. Hence, to express that c_2 was merged into c_1, we set $U[c_2] := [\$a \mapsto \$x, \$b \mapsto \$y] * c_1$. Afterwards $\text{find}(c_1[\$x, \$y])$ remains as-is, but $\text{find}(c_2[\$a, \$b])$ returns $c_1[\$a, \$b]$.

3.3 Equivalence in Slotted E-Graphs

As slotted e-graphs represent sets of terms, we also need to extend the equivalence relation $\approx$ on terms to e-graphs. For this we define an equivalence relation $\cong$ over e-nodes, so that $n \cong n'$, if and only if the equivalence of n and n' can be proven using the rules defined in Figure 6. The key difference between e-nodes and terms is that they recurse using renamed ids, instead of recursively containing terms. Thus, we need an equivalence relation ($\equiv$) for the renamed ids first:

Definition 6 (Equivalence)**.** We define a relation $\equiv$ on renamed ids as follows: $m_1 * a_1 \equiv m_2 * a_2$, iff $\text{find}(m_1 * a_1) = m_1' * c$ and $\text{find}(m_2 * a_2) = m_2' * c$, with $m_2'^{-1} * m_1' \in M[c].G$. Similarly, we define a relation $\cong$ between e-nodes as follows:

$$\$x \cong \$x \qquad m * a \cong m' * a', \text{ if } m * a \equiv m' * a'$$

$$f(x) \cong f(y), \text{ if } x \cong y \qquad x_1, \ldots, x_k \cong x_1', \ldots, x_k', \text{ if } x_i \cong x_i' \text{ for all } i$$

$$\text{bind } \$s_1\ x \cong \text{bind } \$s_2\ y, \text{ if } (\$s_1\ \$f) * x \cong (\$s_2\ \$f) * y, \text{ where } \$f \text{ fresh}$$

In order to detect whether two e-classes contain e-nodes that are equal up to renaming, we compute a name-independent normal form of e-nodes, called *shapes*.

Definition 7 (Shapes)**.** The e-node n' is the *shape* of an e-node n, if there exists a renaming m, s.t.

(1) $n \cong m * n'$ and $\text{dom}(m) = \text{slots}(n')$
(2) all renamed ids $m' * a'$ occurring in n' are leaders, i.e. $\text{find}(m' * a') = m' * a'$.
(3) n' is lexicographically minimal under these constraints (slots have a lexicographical ordering starting with $\$0, \$1, \$2, \ldots$)

The key property of shapes is that:

$$\text{shape}(n) = \text{shape}(n'), \text{ if and only if there exists } m, \text{ s.t. } n \cong m * n'$$

Example 3 (A shape decomposition)**.** The e-node $f(c_1[\$x, \$y], c_2[\$y, \$z])]$ can be decomposed into its shape as follows:

$$[\$0 \mapsto \$x, \$1, \mapsto \$y, \$2 \mapsto \$z] * f(c_1[\$0, \$1], c_2[\$1, \$2])$$

assuming that $c_1[\$x, \$y]$ and $c_2[\$y, \$z]$ are leaders, and cannot be further canonicalized.

We will go into more detail about how to compute shapes, later in this section.

The slotted e-graph represents, just like a conventional e-graph, an equivalence relation over terms. We can now finally introduce a relation describing which terms are contained in which equivalence class: Recall that e-nodes and terms have equivalent definitions, except that terms recurse by containing subterms, whereas e-nodes recurse using renamed ids. Thus, an e-node n *represents* a term t, if the renamed ids of n represent the subterms of t, and n and t are otherwise equal. Similarly, a renamed id $m * a$ represents a term t, if $m * a$ represents an e-node n which in turn represents t. For this, we need to define representation for e-nodes too:

Definition 8 (Representation of e-nodes)**.** We say an renamed id $m * a$ represents e-node n, if

(1) there exists a leader $m' * a'$ with $m' * a' \equiv m * a$,
(2) $n' := shape(n)$ satisfies $H[n'] = a'$, and
(3) there exists $m'' \supseteq m'$, s.t. $n \cong m'' * M[a'].B[n'] * n'$

We do not require $m' * a' = \text{find}(m * a)$, so that symmetry-equivalent leaders to $\text{find}(m * a)$ are not excluded. Further, $m'' \supseteq m'$ allows an arbitrary renaming for the redundant slots.

This definition entails that if $m * a$ is leader that represents an e-node n, then $H[\text{shape}(n)] = a$. And conversely, if $H[\text{shape}(n)] = a$, then there exists m, s.t. $m * a$ represents n.

3.4 Enforcing Slotted E-Graphs Invariants

So far we have discussed slotted e-graphs in a relatively static fashion: e.g. how they represent equivalent terms, but not how we extend them with new equivalences, nor why this is correct. The rest of this section discusses the process of merging in slotted e-graphs, and the invariants that we use for that.

A key invariant of a slotted e-graph is that if two renamed ids $m_1 * a_1$ and $m_2 * a_2$ both represent the same e-node n, then they have already been merged ($m_1 * a_1 \equiv m_2 * a_2$). To enforce this invariant, we have the following rebuilding procedure:

$$\text{if } m_1 * a_1 \not\equiv m_2 * a_2 \text{ both represent the same e-node } n,$$
$$\text{then merge them using } \text{union}(m_1 * a_1, m_2 * a_2)$$

Calling $\text{union}(m_1 * a_1, m_2 * a_2)$ guarantees that $m_1 * a_1 \equiv m_2 * a_2$ holds afterwards. Further, union merges e-classes together, preventing the e-node from being stored unnecessarily often in memory. We make use of the hashcons H to detect whether two distinct renamed ids represent the same e-node. Whenever an e-node n is represented by a leader $m_1 * a_1$, then $H[\text{shape}(n)] = a_1$. Hence, if the e-class of some other leader a_2 changes so that $m_2 * a_2$ now also represents n, it tries to overwrite the hashcons with $H[\text{shape}(n)] = a_2$. At this point, we check whether there has already been a previous value in $H[\text{shape}(n)]$, which means we have found a collision, and we can merge both e-classes.

Shape Computation. Every e-node n can be decomposed into its shape and a renaming, s.t. $n \cong m * n'$. But how do we compute the renaming m and the shape n'? First, we canonicalize all renamed ids in n using the find function of the union-find. In conventional e-graphs, this is the only step required to canonicalize an e-node. In slotted e-graphs, however, we also need to canonicalize the slot names.

To perform the canonicalization of slot names, we consider the syntactic representation of n, where we use the $c[\$x, \$y, \ldots]$ syntax for renamed ids. We iterate over the slots of this representation, left-to-right, s.t. whenever we see a slot that we haven't seen before, we map all occurrences of it to the lowest possible unused slot from \$0, \$1, \$2, ... This way, we guarantee that, independent of the original slot names, the canonicalized slots are lexicographically minimal, as required in Definition 7. As an example, let us consider the e-node $n = let(c_1[\$x, \$y], \text{bind } \$z\ c_2[\$z, \$y])$ that would be decomposed into $m * n' = [\$0 \mapsto \$x, \$1 \mapsto \$y] * let(c_1[\$0, \$1], \text{bind } \$2\ c_2[\$2, \$1])$.

Shape Computation Respecting Symmetries. We need to take special care of symmetries when computing shapes. As mentioned before, canonicalizing renamed ids using the unionfind does not find a unique leader, in case the leader has symmetries. Thus, it could be that the leader returned to us by unionfind does not result in the lexicographically minimal e-node, which we require to compute the shape.

To demonstrate this problem, we use the example from above. Assume that $c_2[\$a, \$b] \equiv c_2[\$b, \$a]$ is an established symmetry. Then the correct shape to compute should be $[\$0 \mapsto \$x, \$1 \mapsto \$y] * let(c_1[\$0, \$1], \text{bind } \$2\ c_2\ [\$1, \$2]\ )$, where the changes compared to the example without the symmetry are highlighted in blue. We can see that the two computed shapes are equivalent, but the second one is lexicographically smaller. Choosing the right symmetries that yield the lexicographically smallest e-node cannot be done greedily after we have already mapped the slots to \$0, \$1, . . ., as it can result in a local minimum.

Finding the shape while respecting symmetries boils down to a finding a canonical (i.e. lexicographically minimal) element in a double coset, as covered in [Butler 1984]. Since the sizes of the groups have not been a bottleneck thus far, in the implementation we currently brute-force over all possible symmetries of the renamed ids contained in our e-nodes, to determine the minimum.

E-Node Collisions Within E-Classes. At the beginning of this section, we described how we can detect whenever two e-classes require to be merged as they represent the same e-node, making use of the hashcons. However, in some cases we require to merge an e-class with a different invocation of itself, resulting in a symmetry. This has to be detected and handled specially.

To understand the problem, consider an e-class id c, with the established symmetry $c[\$x, \$y] \equiv c[\$y, \$x]$ and another e-class $d[\$x, \$y]$ containing the e-node $i(c[\$x, \$y])$. By congruence, the symmetry $c[\$x, \$y] \equiv c[\$y, \$x]$ should entail $i(c[\$x, \$y]) \cong i(c[\$y, \$x])$ and thus $d[\$x, \$y] \equiv d[\$y, \$x]$. However, this equation cannot be found purely using the hashcons.

To solve this issue, we check whenever we recompute the shape of an e-node, whether this shape establishes another symmetry. Recall that we iterate over all possible symmetries in the shape computation. Whenever we detect that two sets of invocations result in the same shape, we derive a symmetry from it. For our example, when computing the shape of $i(c[\$x, \$y])$. We would consider both $i(c[\$x, \$y]) \cong [\$0 \mapsto \$x, \$1 \mapsto \$y] * i(c[\$0, \$1])$ and the permuted form $i(c[\$y, \$x]) \cong [\$0 \mapsto \$y, \$1 \mapsto \$x] * i(c[\$0, \$1])$. And as we notice that both yield the same shape, we have detected another symmetry $[\$0 \mapsto \$x, \$1 \mapsto \$y] * [\$0 \mapsto \$y, \$1 \mapsto \$x]^{-1} = [\$x \mapsto \$y, \$y \mapsto \$x]$ for d.

3.5 Merging Slotted E-Classes

In this section we explain how — and why — the union operation works in a slotted e-graph. After performing the operation $\text{union}(m_1 * a_1, m_2 * a_2)$, the equivalence $m_1 * a_1 \equiv m_2 * a_2$ is recorded by the graph. Depending on how it is called, the function establishes redundancies, symmetries, or merges e-classes. Assuming that $m_1 * a_1$ and $m_2 * a_2$ are already canonicalized by the union-find, the *union* procedure consists of three steps:

1. Establish redundant slots. If $\text{slots}(m_1 * a_1) \neq \text{slots}(m_2 * a_2)$, we establish redundant slots. Concretely, this means that the slot set $M[a_1].S$ shrinks so that it becomes a subset of $slots(m_1^{-1} * m_2 * a_2)$, and $M[a_2].S$ analogously. Note that whenever a slot is marked redundant, then also all slots in the same orbit of the permutation group needs to be marked as redundant. Or in short, if you can swap \$x with \$y by a symmetry, then if \$x is redundant also \$y needs to be redundant.

2. Add a group symmetry. If the two ids coincide, i.e. $a_1 = a_2 = a$ for some a, then we add a group symmetry to the e-class id a. For example, if $slots(a) = \{\$x, \$y\}$ and we union $a[\$x, \$y]$ with $a[\$y, \$x]$, then we add the symmetry $[\$x \mapsto \$y, \$y \mapsto \$x]$ to the permutation group of a. In general, the resulting permutation is $m_1^{-1} * m_2$.

3. Merge e-classes. If the two ids differ, i.e. $a_1 \neq a_2$, then we do "classical merging" of two e-classes, where one e-class gets merged into the other. Note that the renaming $m_1^{-1} * m_2$ tells us which slots in a_2 correspond to which slots in a_1, so that we can merge them accordingly. We set

$U[a_2] := m_1^{-1} * m_2 * a_1$ to make a_2 a follower of a_1, while respecting the renaming between their slots. When merging a_2 into a_1, we move all the shapes from one a_2 to a_1, leaving a_2 empty. We also merge the groups of both e-classes (by unioning their generators). After all, the two e-classes are now equal, thus each symmetry established in one e-class holds for the other e-class too.

To uphold our invariants, we re-compute the shapes of all e-nodes which reference *changed* e-classes, after union completed. An e-class *changed*, if (1) it lost a slot (a redundancy was established), or (2) it gained a symmetry, or (3) it was merged into another e-class, and is now a follower.

3.6 E-Matching

Finally, we discuss e-matching for slotted e-graphs. For this we use an altered version of the "abstract algorithm" presented in Figure 1 of De Moura and Bjørner [2007]. We use the following pattern language, where extensions of ground patterns in e-graphs are marked in green:

$$\begin{aligned}
&\text{pattern variables } x \\
&\text{pattern children } pc ::= p \mid pc_1, \ldots, pc_k \mid \text{bind } \$x\ pc \mid \$x \qquad k \geq 1 \\
&\text{patterns } \ p ::= f \mid f(pc) \mid x
\end{aligned}$$

We say that an e-node n *matches* a pattern p, if they are syntactically equal after replacing all renamed ids of n, and all subpatterns of p, by a constant, $\bot$. We define the function $match(p, m * a, \beta, m_p)$ where

- p is the pattern, and $m * a$ the renamed id we want to match against
- β is a *substitution* mapping from pattern variables to renamed ids, and
- m_p is a renaming that maps the slots from the e-graph to the slots of the pattern.

The *match* algorithm is initially called with $\beta = m_p = \emptyset$, and where m is the identity renaming from $slots(a)$ to itself. The *match* function returns a set of "matches" (pairs of (β, m_p)).

The algorithm works as follows:

$$match(x, m * a, \beta, m_p) = \begin{cases} \{(\beta \cup \{x \mapsto m * a\}, m_p)\}, & \text{if } x \notin \text{dom}(\beta), \\ \{(\beta, m_p)\}, & \text{if } \beta(x) \equiv m * a, \\ \emptyset, & \text{otherwise} \end{cases}$$

$$match(f, m * a, \beta, m_p) = \begin{cases} \{(\beta, m_p)\}, & \text{if } m * a \text{ represents } f, \\ \emptyset, & \text{otherwise} \end{cases}$$

$$match(f(pc), m * a, \beta, m_p) = \bigcup_{f(nc) \text{ is represented by } m*a} S_k,$$

where $m_1 * a_1, ..., m_k * a_k$ are the renamed ids in $f(nc) - p_1, ..., p_k$ are the subpatterns in $f(pc)$, m'_p is the minimal renaming, s.t. $m'_p * f(nc)$ matches $f(pc)$, and where $m_p \cup m'_p$ is a bijective renaming (if no such m'_p exists, set $S_k := \emptyset$), and where we define S_i recursively as follows:

$$\begin{aligned}
S_0 &= \{(\beta, m_p \cup m'_p)\} \\
S_{i+1} &= \bigcup_{(\beta^*, m_p^*) \in S_i} match(p_{i+1}, m_{i+1} * a_{i+1}, \beta^*, m_p^*)
\end{aligned}$$

Technically, if a contains an e-node with a redundant or bound slot, it would represent infinitely many e-nodes, due to infinitely many slots we could rename that slot to. However, for the purpose of this algorithm, it suffices to pick any fresh slot for them.

3.7 Summary

In this section, we have introduced slotted e-graphs formally. Slotted e-graphs represent terms that have variables — bound or free. We showed how this extends a congruence relation to include renamings of variables, and how we need to extend the e-graph data structures to take these renamings into account. Finally, we discussed why slotted e-graphs work, in particular, which invariants they uphold and how merging and e-matching works in the context of slots and renamings.

4 Evaluation

We implemented the slotted e-graph data structure in an open-source Rust library called `slotted`. With `slotted`, users of languages with variables can perform equality saturation by: (1) defining the term language, which is easy, as variables and binders can directly be represented via *slots* - no special encoding of variables is required; (2) defining rewrite rules, which is easy, as `slotted` allows writing rules using familiar notation without having to worry about naming collisions, provides a built-in mechanism to check if a slot is free in a term, and dedicated syntax for substitution - no shifting or renaming rules are required; (3) performing equality saturation by initializing a slotted e-graph, growing it by applying rewrites, and extracting from it.

In this section, we first demonstrate how `slotted` is used for rewriting in a simple array language and systematically evaluate the benefits in terms of ease-of-use and memory efficiency compared to `egg`, the state-of-the-art e-graph implementation for equality saturation.

Then, we present two practical case studies from the domains of theorem proving and compiler optimizations that demonstrate the benefits of using slotted e-graphs in practice.

4.1 Using `slotted` for Rewriting in a Functional Array Language

`slotted` enables users to define their own language and rewrite rules while relying on an efficient, built-in support for binders. We demonstrate this by comparing how a simple functional array language is rewritten using `slotted`, versus how it is rewritten using `egg` with a de Bruijn encoding of variables, as presented in [Kœhler et al. 2024].

Our evaluation example is to optimize functional array code by exploring different ways to fuse or fission operators, trading-off memory usage and other performance aspects, such as redundant computations. We might want to perform the following program transformation:

$$\lambda f_1.\ \lambda f_2.\ \lambda f_3.\ \lambda f_4.\ \lambda y.\ \mathit{map}\ (\mathit{map}\ (\lambda x.\ f_4\ (f_3\ (f_2\ (f_1\ x))))))\ y \qquad \longmapsto \tag{A}$$

$$\lambda f_1.\ \lambda f_2.\ \lambda f_3.\ \lambda f_4.\ \lambda y.\ \mathit{map}\ (\mathit{map}\ (\lambda x.\ f_4\ (f_3\ x)))\ (\mathit{map}\ (\mathit{map}\ (\lambda x.\ f_2\ (f_1\ x)))\ y) \tag{B}$$

The initial program (A) applies functions f_1, ..., f_4 to each element of a two-dimensional matrix y using two nested *map*s. The alternative program (B) introduces an intermediate matrix to store the result of first applying f_1, f_2 before applying f_3, f_4. The alternative program might be preferable, for example, in a context where the result of applying f_1, f_2 to y is reused elsewhere. (B) can be derived from (A) by applying the following rewrite rules in the correct order:

$$(\lambda x.\ b)\ e \longmapsto b[x := e] \qquad (\beta\text{-reduction})$$

$$\lambda x.\ f\ x \longmapsto f \quad \text{if } x \text{ not free in } f \qquad (\eta\text{-reduction})$$

$$\mathit{map}\ f\ (\mathit{map}\ g\ y) \longmapsto \mathit{map}\ (\lambda x.\ f\ (g\ x))\ y \qquad (\text{map-fusion})$$

$$\mathit{map}\ (\lambda x.\ f\ gx) \longmapsto \lambda y.\ \mathit{map}\ f\ (\mathit{map}\ (\lambda x.\ gx)\ y) \quad \text{if } x \text{ not free in } f \qquad (\text{map-fission})$$

This example is particularly interesting to highlight the benefits of `slotted`, because all 4 rewrite rules manipulate binders, in particular eliminate or introduce new binders. Let us first look at how easy it is to define our language and rewrite rules using `slotted`.

```
define_language! {
 pub enum ArrayLang {
   // lambda calculus:
   Lam(Bind<RenamedId>),
   App(RenamedId,
       RenamedId),
   Var(Slot),
   Let(RenamedId,
       Bind<RenamedId>),

   // rest:
   Number(u32),
   Symbol(Symbol),
 }
}
```

```
pub fn rules() -> Vec<Rewrite<ArrayLang>> { vec![
  // lambda calculus:
  rw!("eta"; "(lam $x (app ?f (var $x)))" => "?f", if !slot_free_in("x", "f")),
  rw!("beta"; "(app (lam $x ?body) ?e)" => "?body[(var $x) := ?e]"),
  // perform substitution explicitly as an alternative to the "[_ := _]" syntax:
  rw!("let-intro"; "(app (lam $x ?body) ?e)" => "(let ?e $x ?body)"),
  rw!("let-unused"; "(let ?t $x ?b)" => "?b", if !slot_free_in("x", "b")),
  rw!("let-var-same"; "(let ?e $x (var $x))" => "?e"),
  rw!("let-app";"(let ?e $x (app ?a ?b))"=>"(app (let ?e $x ?ae) (let ?e $x ?b))",
    if or(slot_free_in("x", "a"), slot_free_in("x", "b"))),
  rw!("let-lam-diff"; "(let ?e $x (lam $y ?body))"=>"(lam $y (let ?e $x ?body))",
    if slot_free_in("x", "body")),
  // map fusion and fission:
  rw!("map-fusion"; "(app (app map ?f) (app (app map ?g) ?arg))" =>
    "(app (app map (lam $x (app ?f (app ?g (var $x))))) ?arg)"),
  rw!("map-fission"; "(app map (lam $x (app ?f ?gx)))" =>
    "(lam $in (app (app map ?f) (app (app map (lam $x ?gx)) (var $in))))",
    if !slot_free_in("x", "f"))
] }
```

Listing 1. `slotted` enables defining languages and rewrite rules while relying on built-in support for binders. Here we define a language and rewrite rules for a functional array language in less than 40 lines.

Ease of Use. Listing 1 shows how we define our language and rewrite rules. A language with variables in `slotted` is defined as an algebraic datatype, with constructors that may refer to slots using the `Slot` type, renamed slotted e-class ids using the `Id` type, or arbitrary data (**`u32`**, `Symbol`).

Defining a purely syntactic rewrite rule in `slotted` only requires defining a left-hand-side and right-hand-side pattern, and optionally providing a side condition for triggering the rewrite. Variables bound inside a pattern are slots (`$x`), while the rest are pattern variables (`?e`). In particular, there is built-in and extensible support for expressing substitution using the `[_ := _]` syntax (see the "beta" rule in Line 18).[4] Testing whether a slot is free inside a pattern variable instantiation is also covered by `slot_free_in` (see the "eta" rule in Line 17). Common subtleties when externally encoding binders in egg are avoided, for example there is no need to worry about name collisions (e.g. `$x` and `$y` are guaranteed to be different in `"let-lam-diff"`), no need to shift any de Bruijn indices, and no need to implement a custom free variable analysis.

All in all, our `slotted` implementation takes less than 35 lines of code. By contrast, in egg, a name based implementation as in [Willsey et al. 2021] takes about 200 lines of code, while a de Bruijn index based implementation as in [Kœhler et al. 2024] takes about 250 lines of code. Both name based and index based implementations are tricky to get right.

Memory Efficiency. We now compare the efficiency of our `slotted` implementation to the efficiency of a de Bruijn encoding in egg (`egg-db`), corresponding to prior work [Kœhler et al. 2024]. We also compare against that same de Bruijn encoding, but instead implemented in `slotted` (`slotted-db`), without using slots. Comparing `slotted-db` to `egg-db` allows observing the overhead and optimization potential of the `slotted` library compared to the egg library when slots are not leveraged.[5] All three implementations perform substitution via rewriting and pushing the let-binding though the expression (e.g. the let-rules in Listing 1). To increase the difficulty of rewriting (A) into (B), we add a varying amount of parameters to every function. By adding 2 parameters, we use $((f_1\ p_1)\ p_2)$ instead of f_1, where the p_i are bound at the top level. We give rewriting a budget of 5 min and 4GB RAM, on an Intel(R) Core(TM) i7-8665U CPU.

Figure 8 shows the results in terms of number of e-nodes, memory consumption and runtime.

[4]The substitution syntax [_ := _] defaults to using term-based substitution, applied on the canonical term of an e-class. However in the benchmarks we have used let-based explicit substitution.

[5]`slotted` does not use any egg code, and was built from scratch.

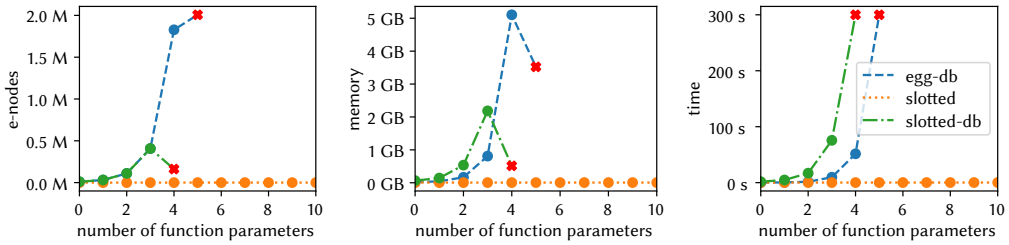

Fig. 8. Number of e-nodes, amount of memory, and runtime to rewrite (A) into (B) with additional function parameters. Unsurprisingly, the de Bruijn encoding in `slotted` (`slotted-db`) performs worse than the de Bruijn encoding in egg (`egg-db`) due to the overhead of slots and missing framework optimizations. Despite that overhead, `slotted` vastly outperforms `egg-db` thanks to the efficiency of slotted e-graphs.

First, we observe that, while `slotted-db` has the same amount of e-nodes as `egg-db`, there is an overhead. For 3 function parameters, `slotted-db` uses 2.7× more memory, and runs 7.8× slower. Part of this overhead is intrinsic to slots, but part of it is also due to the fact that egg is highly optimized compared to `slotted`, as we tried to keep our implementation simple (e.g. we did not consider cache-friendliness, deferred rebuilding or compiled e-matching).

Second, we observe that both de Bruijn implementations fail when increasing parameter count, with an exploding amount of e-nodes, memory and runtime, while the `slotted` implementation scales very well, with a consistently low amount of e-nodes, memory, and runtime. The downfall of de Bruijn is its need to constantly shift indices, for example, fissioning one function and its parameters triggers rewrites like the following:

$$map\ (\lambda x.\ f\ (((f_1\ p_1)\ \dots\ p_N)\ x)) \longmapsto \lambda y.\ map\ f\ (map\ (\lambda x.((f_1\ p_1)\ \dots\ p_N)\ x))\ y$$

Which triggers a shift when using de Bruijn indices:

$$map\ (\lambda.\ f\ (((\hat{f_1}\ \hat{p_1})\ \dots\ \hat{p_N})\ \hat{0})) \longmapsto \lambda.\ map\ f\ (map\ (\lambda.\ \uparrow_1^1\ \ (((\hat{f_1}\ \hat{p_1})\ \dots\ p_N)\ \hat{0})))\ \hat{0}$$

Here, $\uparrow_1^1$ adds 1 to all indices at or above 1, which requires $N + 1$ rewrites to propagate through, copying huge parts of the e-graph as differently shifted e-classes are not considered equivalent:

$$\uparrow_1^1\ \ (((\hat{f_1}\ \hat{p_1})\ \dots\ p_N)\ \hat{0}) \longmapsto {}^{(N+1)}((f_1\ \hat{+}\ 1\ p_1\ \hat{+}\ 1)\ \dots\ p_N + 1)\ \hat{0}$$

Both `slotted-db` and `egg-db` exceed our 4GB budget using only 4 function parameters. Due to the need for shifting rewrites, 14 iterations of equality saturation are required by `egg-db` to achieve the goal, which takes 1827k e-nodes, 922k e-classes, 5.1GB RAM, and 52s to achieve the rewrite goal.

In comparison, `slotted` maintains congruence modulo renaming, avoiding these $N + 1$ rewrites, and maintaining sharing instead of duplicating e-classes purely to shift (rename) variables. Only 6 iterations of equality saturation are required independently of the number of function parameters, requiring at most 214 e-nodes (4 orders of magnitude less), 95 e-classes, 6MB RAM (3 orders of magnitude less), and 0.22s (1 order of magnitude less).

This experiment demonstrates that, even for such a relatively simple (albeit constructed) rewrite goal, slotted e-graphs are orders of magnitude more efficient than an encoding with de Bruijn indices, thanks to their built-in support for binders and congruence modulo renaming. The following case studies demonstrate that this efficiency also translates to benefits in real workloads.

4.2 Using `slotted` for Sparse Tensor Compilation with SDQL

SDQL [Shaikhha et al. 2022] is an intermediate language that can express database queries and tensor algebra expressions in a unified manner. The central data structure behind SDQL is a semi-ring dictionary, which can subsume collection types such as sets/bags (used in databases) and arrays/tensors (used in tensor algebra). Furthermore, the algebraic nature of SDQL allows loop optimizations to be expressed as local algebraic rewrite rules. For example, the following rewrite rule has the same effect as the vertical loop fusion:

```
sum(<k1, v1> in                          sum(<k2, v2> in R)
  sum(<k2, v2> in R) {k2 -> body1})  ⇝     let <k1, v1> = <k2, body1> in
  body2                                   body2
```

The original program creates an intermediate semi-ring dictionary, which is immediately consumed. The optimized program removes this allocation by fusing the loops. Vertical loop fusion is a crucial optimization in data processing systems, including DB and ML systems [Shaikhha et al. 2022].

Storel [Schleich et al. 2023] uses equality saturation on top of SDQL to compile tensor algebra computations over sparse data formats. As opposed to sparse tensor compilers such as TACO [Kjolstad et al. 2017], where the computation and data format are separated, Storel unifies them by expressing both as SDQL expressions. The tensor computation is expressed as computation over semi-ring dictionaries. The sparse data format specification is expressed as an SDQL program, converting the physical data layout to semi-ring dictionaries. The Storel compiler uses two passes of equality saturation: the first pass only optimizes the tensor computation, and the second pass fuses the data format specification with the optimized tensor computation.

Due to the heavy usage of binders, the Storel system relies on the de Bruijn encoding for egg by Koehler [2022]. It relies on 44 rewrite rules consisting of semi-ring algebraic simplifications, loop fusion, loop-invariant code motion, normalization rules, and let-binding rules.

Implementing SDQL over Slotted E-Graphs. We reimplement Storel based on slotted e-graphs. We generalize the original set of rewrite rules to a more fine-grained rule set. The fine-grained rule set also uses 44 rewrite rules, however, the rewrite rules are simpler and more general than the ones in the original coarse-grained rule set. There are rewrite rules in the original coarse-grained set in Storel that can be derived from more fine-grained rules. For example, consider the following rewrite rule, which removes an intermediate semi-ring dictionary:

```
(sum(<k, v> in R) {k -> body})(idx)  ⇝  let k = idx in let v = R(k) in body
```

This rewrite rule can be recovered by composing the following set of more general rewrite rules:

```
(sum(<k, v> in R) {k -> body})(idx)
  ⇝  sum (<k1,v1> in (sum(<k,v> in R) {k->body})) if(k1 == idx) v1
  ⇝  sum(<k, v> in R) let <k1,v1> = <k,body> if(k == idx) v1
  ⇝  sum(<k, v> in R) if(k == idx) body  ⇝  let k = idx in let v = R(k) in body
```

Furthermore, this opens up opportunities to compose these fine-grained rules to create complicated rewrites that are missing from the coarse-grained ones. However, these more basic rewrite rules lead to a larger search space. We report the results for the fine-grained rule sets that make the scalability more challenging.

Performance of Slotted E-Graphs. Table 1 shows the compilation metrics for `slotted` compared with egg. We set a limit on memory usage, terminating equality saturation when the memory usage exceeds 1.5GB. In all cases, both systems find the best program, except for the second phase of MTTKRP, as will be discussed later.

Kernel	System	Iters.	Nodes	Classes	Saturated	Memory (MB)
ΣMMM (1st)	egg	10	84	34	✓	1.77
	slotted	5	30	15	✓	2.69
MTTKRP (1st)	egg	18	2,240	300	✓	4.34
	slotted	8	466	59	✓	5.89
MMM (1st)	egg	18	673	151	✓	2.77
	slotted	9	130	27	✓	3.61
TTM (1st)	egg	17	2,655	298	✓	3.36
	slotted	9	280	47	✓	4.05
BATAX (1st)	egg	148	1,401,722	1,108,225	✗	1570.02
	slotted	13	97,238	16,767	✗	1705.47
ΣMMM (2nd)	egg	49	14,334	2,890	✓	14.92
	slotted	15	314	75	✓	3.94
MTTKRP (2nd)	egg	424	2,970,959	378,263	✗	1691.17
	slotted	25	4,891	307	✓	75
MMM (2nd)	egg	252	287,536	21,582	✓	209.91
	slotted	18	1,308	128	✓	18.72
TTM (2nd)	egg	421	2,641,070	496,441	✗	1540.89
	slotted	25	2,044	196	✓	29.98
BATAX (2nd)	egg	394	2,386,580	506,848	✗	1833.28
	slotted	12	71,643	10,435	✗	1585.77

Table 1. Compilation metrics reported by egg and slotted. Both systems find the best program in all cases (except for MKTTKRP as can been in Table 2).

We observe that the 1st phase of the compiler in most kernels saturates, and both systems consume a similar amount of memory. However, the number of e-nodes and e-classes is significantly less for slotted e-graphs. The situation for the 2nd phase of the compiler is different. Slotted e-graph saturates for all kernels except for the BATAX kernels, due to the extensive use of associativity rules. However, egg cannot saturate for the TTM and MTTKRP kernels, despite using an order of magnitude more memory and three orders of magnitude more e-nodes and e-classes.

The more fine-grained set of rewrite rules makes the compiler more general. Furthermore, the specialized coarse-grained rules (such as the one expressed above) can be recovered by the fine-grained rules. However, this recovery leads to a larger search space and can result in not extracting the optimized program given a memory/time budget.

Table 2 shows the comparison of egg and slotted for the second phase of the MTTKRP kernel. For completeness, here we also include both the coarse-grained rule sets used in Storel. When using the fine-grained set of rewrite rules, the egg framework fails to saturate and cannot find the

System	Rewrite Space	Iters.	Nodes	Classes	Saturated	Memory (MB)	Best Found (Iter #)
egg	Fine-grained	424	2,970,959	378,263	✗	1691.17	✗
	Coarse-grained	419	1,256,975	76,589	✓	1000.48	✓ (18)
slotted	Fine-grained	25	4891	307	✓	75	✓ (12)
	Coarse-grained	21	2316	205	✓	26.98	✓ (12)

Table 2. Compilation metrics reported by egg and slotted for the second phase of MTTKRP based on the coarse- and fine-grained set of rewrite rules.

optimized program. However, with the coarse-grained set of rewrite rules, given the smaller search space, egg successfully saturates and finds the optimized program. In particular, for coarse-grained rewrite rule set, it requires 20× fewer iterations to saturate. `slotted` finds the best program after 12 iterations for both rewrite sets, whereas egg needs 18 iterations for the coarse-grained rewrite space. Furthermore, `slotted` successfully saturates in both cases, requiring two orders of magnitude less memory and three orders of magnitude less e-nodes and e-classes.

4.3 Using `slotted` for Theorem Proving with Lean

Interactive theorem proving provides an environment in which users construct mathematical proofs. Users are supported by the computer, which checks that proofs are indeed correct and assists by providing capabilities for constructing some proofs automatically. Theorem proving is an interesting use case for equality saturation, as it does not aim to optimize an initial term, as we did in the prior case study. Instead, it reasons about the equality of two terms.

Kœhler et al. [2024] integrated the equality saturation engine egg for use as a proof tactic in the Lean theorem prover [de Moura and Ullrich 2021]. Notably, this proof tactic omitted important fragments of Lean's expression language, including all forms of binders. Rossel [2024] extended the supported fragment to include binders, aiming to address two necessary considerations:

(1) Which (definitional) equality or reduction rules relating to binders are built into Lean's type theory, and how to encode them in an e-graph?
(2) Which problems arise during syntactical rewriting over expressions containing de Bruijn variables, and how to detect and resolve them during equality saturation?

To address (1), rewrites which encode Lean's only two definitional equality rules for binders were implemented: β-reduction and η-reduction [Carneiro 2024].[6] In de Bruijn representation, these are:

$$(\lambda e)\ a \ \to_\beta \ \downarrow(e[\hat{0} := \uparrow a]) \qquad\qquad \lambda(e\ \hat{0}) \ \to_\eta \ \downarrow e \quad \text{if } \hat{0} \notin \mathit{fvars}(e)$$

β-reduction requires shifting de Bruijn indices (written as $\downarrow$ and $\uparrow$) as well as a substitution operator (written as $[\cdot := \cdot]$). The simpler η-reduction only requires a shifting operator, but is also contingent upon a precondition which requires the ability to query the set of free variables.

To address (2), checks are added in every rewrite rule to detect and, if possible, resolve problems which can occur during syntactic rewriting of expressions containing de Bruijn variables. A widely known example of such a problem is *invalid capture* of variables during substitution. Other problems of this kind are referred to as *Invalid Matching Problems (1) and (2)* in [Rossel 2024], and a problem called *Invalid Non-Matching* discussed in [Koehler 2022].

In summary, rewriting Lean expressions in de Bruijn representation in egg requires introducing non-trivial machinery to represent definitional equality rules (Consideration 1), as well a variety of checks and corrections to be performed on all rewrites (Consideration 2).

In this case study, we have thus extended the open-source proof tactic with a new alternative backend based on the `slotted` implementation, as sketched in Figure 9.

[6] This is a slight simplification and based on Lean's *algorithmic* definitional equality relation.

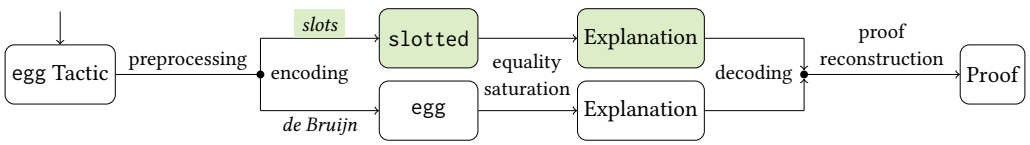

Fig. 9. Overview of our integration of `slotted` into the egg proof tactic.

Problem	Slotted	De Bruijn	Named
Invalid Capture	automatic	manual	manual
Invalid Matching (1)	automatic	manual	manual
Invalid Matching (2)	manual (simple)	manual (complex)	manual (complex)
Invalid Non-Matching	automatic	manual	manual
Free Variable E-Class Analysis	not required	required	required
Shifting/Renaming Operation	not required	required	required
Substitution Operator	required	required	required

Table 3. Comparison of the implementation complexity for handling Lean expressions with binders.

Implementation Complexity. An immediate benefit we observe by using slotted e-graphs is a significant reduction in implementation complexity for handling binders correctly. Table 3 lists the aforementioned problems encountered when using binders in e-graphs and compares the effort required to address them for de Bruijn and named representations in e-graphs, versus slotted.

We observe that most problems simply cannot occur in slotted e-graphs and thus require no attention. The notion of a shifting/renaming operation is not required, simplifying the implementation of β- and η-reduction. There is no need for an e-class analysis tracking free variables. And while an explicit substitution operator is still required, we simply use the one provided by `slotted`.

Where slotted e-graphs do not automatically solve a problem, they still simplify manual implementation. One example of this is addressing the Invalid Matching Problem (2), which occurs when the rewrite $(\lambda.\ ?x)\ c \Rightarrow ?x$ is applied to $(\lambda.\ \hat{0})\ c$, producing the invalid open term $\hat{0}$. The underlying issue is that pattern variables are not allowed to match against variables which are introduced by binders in the rewrite's pattern. In [Rossel 2024] this is addressed in `egg` by a non-trivial traversal of a rewrite's pattern while tracking positions and depths of binders to match them to de Bruijn indices. In slotted e-graphs we can address this problem directly as slots are comparable without knowledge of their binder. If P denotes the set of slots appearing in the rewrite's pattern and $S(\sigma)$ denotes the slots appearing in the renamed identifiers contained in the substitution σ, then σ is a valid if P and $S(\sigma)$ are disjoint. That is, in our implementation, we simply check $P \cap S(\sigma) = \emptyset$.

Efficacy of Slotted E-Graphs for Real-World Examples. We evaluate the efficacy of slotted e-graphs for automated equational reasoning in Lean by proving real-world theorems using both `egg` and `slotted` as backends to the extant `egg` Lean proof tactic. For this purpose, we consider theorems from the Lean mathematics library Mathlib [mathlib Community 2020]. To identify candidate theorems, we override the behavior of Lean's simplifier tactic `simp only` to call the `egg` tactic under the hood, while checking that we close the current proof goal, which we require to be an equality. Given these constraints, we obtain 492 test cases, which we run with a 30-second timeout. We eliminated 65 cases which failed in the Lean pre- or post-processing. Table 4 compares `egg` and `slotted` on the remaining 427 cases, of which both were able to successfully prove 290, about 68%.

Backend	Successful Proven Theorems	Average Explanation Length	...With Binders
`slotted`	290	6.2	6.7
`egg`	290	5.4	12.7

Table 4. Comparison of the `egg` and `slotted` backends on 427 simple test cases from Mathlib.

Considering the successful test cases, 30 involved rules relating to binders (β-, η-reduction, shifting, substitutions) in their resulting explanation. For these test cases, we list the explanation length metric separately. We find that both backends manage to prove the exact same theorems, with slotted e-graphs requiring 9% fewer e-nodes on average for tests involving binders. We also see a clear effect on explanation length when binders are involved, with slotted e-graphs producing 47% shorter explanations on average. This can be attributed to the fact that explanations for e-graphs using de Bruijn indices include pervasive uses of variable shifts.

While increased explanation length does not pose a problem for the simple `simp only` test cases, it can do so for more complex theorems. Consider the following example from Mathlib – a theorem from order theory characterizing sup-prime elements in semilattices:

```
theorem not_supPrime : ¬SupPrime a ↔ IsMin a ∨ ∃ b c, a ≤ b ⊔ c ∧ ¬a ≤ b ∧ ¬a ≤ c
  := by egg [not_forall, not_exists, ..., SupPrime]
```

In Mathlib, this theorem is solved by combining `push_neg`, a dedicated tactic for pushing negations into logical connectives, with specific rewrites about the concrete mathematical objects (here, sup-prime elements). We can prove this theorem using equality saturation. While the `egg` backend succeeds, it produces an explanation with 4657 proof steps, which takes the Lean tactic's proof reconstruction algorithm over 15 minutes to process before failing. Notably, 3793 (81%) of these steps are only concerned with applying and propagating shifts of de Bruijn variables. In comparison, **the `slotted` backend succeeds in under 1 second**, with an explanation of just 14 steps.

An even bigger difference occurs when trying to prove this theorem, also from order theory:

```
theorem not_supIrred : ¬SupIrred a ↔ IsMin a ∨ ∃ b c, b ⊔ c = a ∧ b < a ∧ c < a := by
  have h x y : x ⊔ y = a ∧ ¬x = a ∧ ¬y = a ↔ x ⊔ y = a ∧ x < a ∧ y < a := by ...
  egg [not_forall, not_exists, ..., SupIrred, exists₂_congr h]
```

With the `egg` backend, we never manage to prove the theorem. After 5 minutes, the e-graph already explodes to 2.6 million e-nodes. Running the example for 3 hours peaked at 40GB of memory usage before crashing. **With `slotted`, we prove this theorem in less than 1 second.**

5 Related Work

E-Graphs and Equality Saturation. E-Graphs were originally proposed as an efficient data structure for maintaining congruence closure [Kozen 1977; Nelson and Oppen 1980]. Egg [Willsey et al. 2021] has re-ignited interest in using the data structure for equality saturation, due to an efficient rebuilding technique and e-class analysis, both crucial for the practical use cases. Equality saturation [Tate et al. 2009] uses the e-graph data structure to quickly explore a set of equivalent terms, which is useful both in optimization and reasoning. Recent projects have explored the use of equality saturation to optimize software [Kœhler et al. 2024; Shaikhha et al. 2022] and hardware [Cheng et al. 2024; Wang et al. 2023] or in theorem provers [Kœhler et al. 2024; Rossel 2024] and verification [Dickerson et al. 2024], only to name a few projects.

Dealing with Variables in E-Graphs. . Conventional e-graphs do not offer any support for representing variables. Therefore, different strategies have been developed to represent variables in e-graphs. Cao et al. [2023] work with a first-order lambda calculus that is designed to avoid having to deal with α-renaming during β-reduction. [Nandi et al. 2020] largely avoid bindings in their language design preferring a combinator-style, similarly, Glenside by [Smith et al. 2021] avoids bound variables and binders altogether. The egg paper represents variables with string names mainly for didactic reasons, and using de Bruijn indices is reported as significantly better by [Kœhler et al. 2024] and [Shaikhha et al. 2022], allowing significantly more complex rewrite problems being solved using the de Bruijn encoding as with string names.

Grechanik [2015] presents a data structure that seems similar to slotted e-graphs for a specialized language, while using it for bisimilarity detection of programs. Their work is restricted to a fixed language with built-in "case" expressions as binders, and cannot express arbitrary binders, as our solution does. We developed slotted e-graphs completely independently of this work.

Dealing with Variables in Term Rewriting and Nominal Rewriting. . Dealing with variables in term rewriting is a long-standing issue. De Bruijn [De Bruijn 1972] suggested their indexing scheme that is still popular today in the implementation of functional languages and in rewriting.

More closely related to our treatment of variables are hierarchical abstract syntax graphs by [Ghica 2021], that provide a graphical representation of bound terms similar to how our slotted e-graph represents sets of terms. However, the closer formal treatment of variables are nominal rewriting techniques [Fernández and Gabbay 2007; Urban et al. 2004] that have developed a formal framework that integrates α-equivalence with binders in first-order syntax.

6 Conclusion

This paper introduces *slotted e-graphs*, an extension of e-graphs, representing terms that differ only in the names of their variables using the same e-node. E-Classes are parameterized by *slots* abstracting over all free variables of the equivalent terms represented by its e-nodes. Referring to an e-class from an e-node relates the free variables from the e-node's context to the e-class' slots.

We have shown, that slotted e-graphs formally maintain an extended congruence relation with slots, that considers terms as equal if they only differ by the names of their variables. The data structure enforces its invariants using a slotted version of union-find and by computing the shapes of e-nodes, a normal form with canonical names for the e-nodes' free variables.

Using our `slotted` implementation, representing languages with variables, such as lambda calculus is greatly simplified, as no dedicated encoding of variables, e.g., using de Bruijn indices is required. Rules are expressed naturally with built-in support for substitution and built-in support for tracking free slots. Our evaluation of two case studies shows, that slotted e-graphs require orders of magnitude less memory than conventional e-graphs for practically relevant rewrite problems and that slotted e-graphs can solve rewrite problems that conventional e-graphs can't.

References

Gregory Butler. 1984. On computing double coset representatives in permutation groups. In *Computational Group Theory: Proceedings of the London Mathematical Society Symposium on Computational Group Theory*, Michael D. Atkinson (Ed.).

David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. 2023. babble: Learning Better Abstractions with E-Graphs and Anti-unification. *Proc. ACM Program. Lang.* 7, POPL (2023), 396–424. doi:10.1145/3571207

Mario Carneiro. 2024. Lean4Lean: Towards a formalized metatheory for the Lean theorem prover. *CoRR* abs/2403.14064 (2024). doi:10.48550/ARXIV.2403.14064 arXiv:2403.14064

Jianyi Cheng, Samuel Coward, Lorenzo Chelini, Rafael Barbalho, and Theo Drane. 2024. SEER: Super-Optimization Explorer for High-Level Synthesis using E-graph Rewriting. In *ASPLOS (2)*. ACM, 1029–1044.

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.

Leonardo De Moura and Nikolaj Bjørner. 2007. Efficient E-matching for SMT solvers. In *Automated Deduction–CADE-21: 21st International Conference on Automated Deduction Bremen, Germany, July 17-20, 2007 Proceedings 21*. Springer, 183–198.

Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *CADE (Lecture Notes in Computer Science, Vol. 12699)*. Springer, 625–635.

Robert Dickerson, Prasita Mukherjee, and Benjamin Delaware. 2024. KestRel: Relational Verification Using E-Graphs for Program Alignment. *CoRR* abs/2404.08106 (2024). doi:10.48550/ARXIV.2404.08106 arXiv:2404.08106

Maribel Fernández and Murdoch Gabbay. 2007. Nominal rewriting. *Inf. Comput.* 205, 6 (2007), 917–965. doi:10.1016/J.IC.2006.12.002

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

Sergei A. Grechanik. 2015. Proving properties of functional programs by equality saturation. *Program. Comput. Softw.* 41, 3 (2015), 149–161. doi:10.1134/S0361768815030056

Johannes Kellendonk and Mark V Lawson. 2004. Partial actions of groups. *International Journal of Algebra and Computation* 14, 01 (2004), 87–114.

Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman P. Amarasinghe. 2017. The tensor algebra compiler. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 77:1–77:29. doi:10.1145/3133901

Thomas Koehler. 2022. *A domain-extensible compiler with controllable automation of optimisations.* Ph. D. Dissertation.

Dexter Kozen. 1977. Complexity of Finitely Presented Algebras. In *STOC*. ACM, 164–177.

Thomas Kœhler, 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.

Lukasz Lachowski. 2018. On the Complexity of the Standard Translation of Lambda Calculus into Combinatory Logic. *Reports Math. Log.* 53 (2018), 19–42. https://rml.tcs.uj.edu.pl/rml-53/02-lachowski.pdf

Serge Lang. 2012. *Algebra*. Vol. 211. Springer Science & Business Media.

Mark V Lawson. 1998. *Inverse Semigroups, the Theory of Partial Symmetries.* World Scientific Publishing Co. Pte. Ltd.

The mathlib Community. 2020. The lean mathematical library. In *CPP 2020*. ACM, 367–381. doi:10.1145/3372885.3373824

Krzysztof Maziarz, Tom Ellis, Alan Lawrence, Andrew W. Fitzgibbon, and Simon Peyton Jones. 2021. Hashing modulo alpha-equivalence. In *PLDI*. ACM, 960–973.

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 *PLDI*. ACM, 31–44.

Greg Nelson and Derek C. Oppen. 1980. Fast Decision Procedures Based on Congruence Closure. *J. ACM* 27, 2 (1980), 356–364. doi:10.1145/322186.322198

Marcus Rossel. 2024. *An Equality Saturation Tactic for Lean.* Master's thesis.

Maximilian Schleich, Amir Shaikhha, and Dan Suciu. 2023. Optimizing Tensor Programs on Flexible Storage. *Proc. ACM Manag. Data* 1, 1 (2023), 37:1–37:27. doi:10.1145/3588717

Ákos Seress. 2003. *Permutation group algorithms.* Number 152. Cambridge University Press.

Amir Shaikhha, Mathieu Huot, and Shideh Hashemian. 2024. A Tensor Algebra Compiler for Sparse Differentiation. In *CGO*. IEEE, 1–12.

Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. 2022. Functional collection programming with semi-ring dictionaries. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–33. doi:10.1145/3527333

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*. ACM, 21–31.

Robert Endre Tarjan. 1975. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)* 22, 2 (1975), 215–225.

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *POPL*. ACM, 264–276.

Christian Urban, Andrew M. Pitts, and Murdoch Gabbay. 2004. Nominal unification. *Theor. Comput. Sci.* 323, 1-3 (2004), 473–497. doi:10.1016/J.TCS.2004.06.016

Zhengrong Wang, Christopher Liu, Aman Arora, Lizy Kurian John, and Tony Nowatzki. 2023. Infinity Stream: Portable and Programmer-Friendly In-/Near-Memory Fusion. In *ASPLOS (3)*. ACM, 359–375.

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 (2021), 1–29. doi:10.1145/3434304

Received 2024-11-14; accepted 2025-03-06