Formal Methods in Computer-Aided Design 2024

Easter Egg: Equality Reasoning Based on E-Graphs with Multiple Assumptions

Eytan Singher and Shachar Itzhaky
Technion - Israel Institute of Technology, Haifa, Israel
{eytan.s,shachari}@cs.technion.ac.il

***Abstract*—E-graphs are a prominent data structure that has been increasing in popularity in recent years due to their expanding range of applications in various formal reasoning tasks. E-graphs allow systematic and efficient treatment of equality, which is pervasive in automated reasoning based on proofs.**

E-graphs handle equality well, but are severely limited in their handling of case splitting and other aspects of propositional reasoning, such as resolution, which introduce branching in provers and solvers. As a consequence, most tools resort to using e-graphs locally, recreating them ad-hoc when they are needed, and then discarding them. In exploratory scenarios, where it is necessary to retain multiple branches simultaneously, this limitation proves to be prohibitive. In particular, in theory exploration—a process where lemmas are discovered and then proven—this poses a significant challenge. Theory exploration must enumerate a space of possible assumptions, and must retain all of them to make progress. This poses a severe limitation on the ability to harness e-graphs for the task.

Our key observation is that in exploratory reasoning tasks, branching represents versions of the same e-graph each with an added assumption, such as "$x > y$" or "is_sorted l". Essentially, each e-graph represents an equality relation, and each branch corresponds to a matching coarsened equality relation. Based on this observation, we present an extension to e-graphs, called *Colored E-Graphs*, as a way to efficiently represent all of the coarsened equality relations in a single structure. A colored e-graph is a memory-efficient equivalent of multiple copies of an e-graph, with a much lower overhead. This is attained by sharing as much as possible between different cases, while carefully tracking which conclusion is true under which assumption. It can be viewed as adding multiple "color-coded" layers on top of the original e-graph structure, representing different assumptions.

We run experiments and demonstrate that our colored e-graphs can support large numbers of assumptions and terms with space requirements that are about $10\times$ lower, and with slightly improved performance.

I. Introduction

E-graphs are a versatile data structure that is used for various tasks of automated reasoning, including theorem proving and synthesis. E-graphs have been popularized in compiler optimizations thanks to their ability to support efficient *rewrites* over a large set of terms, while keeping a compact representation of all possible rewrite outcomes. This mechanism is known as *equality saturation*. It provides a powerful engine that allows a reasoner to generate all equality consequences of a set of known, universally quantified, equalities. Possible uses include selecting the best equivalent of an expression according to some desired metric, such as run-time efficiency [29], size [10], [22], or precision [23] (when used as a compilation phase) and a generalized form of unification, called e-unification, for application of inference steps (when used for proof search).

In this work we focus on a stepping stone for what we address as *exploratory reasoning*: a range of tasks including all the above optimization procedures, as well as theory exploration [26], rewrite rule inference [20], and proof search [16], [5], [14]. Exploratory reasoning, in general, can be thought of as any reasoning task navigating a large space of potential goals or sub-goals that need to be selected based on some criteria. Our motivating example comes from TheSy and Ruler, both of which are theory exploration systems based on e-graphs. A theory exploration system attempts to both discover and prove mathematical properties from a set of definitions and known lemmas. Most of the difficulty in theory exploration comes from the generation and filtering of candidates, rather then from the proof procedure itself. TheSy does so by efficiently filtering a large set of potential conjectures using e-graphs for equality reasoning, and evaluating which should be potentially proved. While e-graphs are effective for equality reasoning [30], handling branching, such as case splitting during proof search, do not have a common solution, and are treated ad-hoc. For example, a special type of node is introduced in [29] to deal with loop conditions, while in [7] a special operator is introduced to reason on expressions under certain contexts, and [26] creates full copies of the e-graph for each branch being explored.

To illustrate this difficulty, we zoom in on an example from theory exploration. As an example scenario, consider trying to discover and prove lemmas on sorted lists: a library containing functions find, is_sorted, and bin_search. We expect to discover lemmas involving these functions; one such lemma might be the property: is_sorted $l \rightarrow$ bin_search $l\,v =$ find $l\,v$. State-of-the-art theory exploration systems [12], [20], [26] have some enumeration strategy over expressions in order to discover candidates. A challenge presents itself when some lemmas in the space require an assumption, in this case is_sorted l. When dealing with e-graphs, adding an assumption would *globally* affect all terms involved in the enumeration, making it impossible to separate conclusions stemming from different assumptions. Because the system cannot know in advance which assumptions will become relevant for discovering equalities, it is required that it also generate and test multiple candidate assumptions. An immediate solution is to create one copy of the graph per assumption, but doing so can significantly increase the memory usage. Moreover, lemmas may depend on

https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_13

one another; for example, $\mathsf{is\_sorted}\, l \to \mathsf{bin\_search}\, l\, v = \mathsf{find}\, l\, v$ depends on transitivity of $\leq$ $(x \leq y \wedge y \leq z \to x \leq z)$. Therefore, just trying the candidates one at a time would mean that the system would prematurely discard candidates depending on the order in which they are tested; alternatively, for each candidate that is validated and becomes a lemma, it would be forced to re-try all the previously failed attempts, which is highly costly.

To overcome this difficulty, we propose an extension of the e-graph data structure. An e-graph naturally represents a congruence relation $\cong$, which is an equality relation over terms (with function applications), which maintains $x \cong y \vdash f(x) \cong f(y)$. The congruence relation is maintained in the e-graph as a set of equivalence classes (e-classes), which can be merged as part of updating the underlying relation. We extend the e-graph data structure into a *Colored E-Graph* to maintain multiple congruence relations at once, where each relation is associated with a color. Our key observation is that each added assumption, can be treated as a new congruence relation, but is only a coarsening of the original relation. The coarsening, then, can be represented as a set of additional merges of e-classes on top of the original e-graph. The main benefit is reducing memory consumption by re-using and sharing most of the e-classes between colors. Going back to the sorted list example, in the colored e-graph there will be a red relation for assuming $x \leq y \wedge y \leq z$, and a blue relation for assuming $\mathsf{is\_sorted}\, l$. Thanks to the size reduction, multiple relations can exist at once, and thus the lemma $\mathsf{is\_sorted}\, l \to \mathsf{bin\_search}\, l\, v = \mathsf{find}\, l\, v$ can be discovered after transitivity of $\leq$ is proven, but without dependency on the order of exploration. Colored e-graphs also support having a hierarchy between different colors, which can benefit from additional sharing of e-classes. For example, the red color representing $x \leq y \wedge y \leq z$ is itself a coarsening of some green color representing just the assumption $x \leq y$.

While the memory footprint for each color is smaller, maintaining the congruence relation and the data structure invariants becomes more challenging. To address this we present specialized data-structure modifications and evaluate them. First, we set up a multi-level union-find where the lowest level corresponds to the root congruence. Second, we change how congruence closure is applied to the individual congruence relations while taking advantage of the sharing between each such relation and the root. Lastly, we present a technique for efficient e-matching over all the relations at once.

Our contributions are:

1) The observation that assumptions induce coarsened e-graphs that share much of the original structure.
2) Algorithms for colored e-graphs operations.
3) Optimizations on top of the basic algorithms to significantly improve resource usage.
4) A colored e-graph implementation, *Easter Egg*[1] and an evaluation that shows an improvement factor in memory usage over the existing baseline, while maintaining similar run-time performance.

[1]https://github.com/eytans/egg/tree/features/color_splits

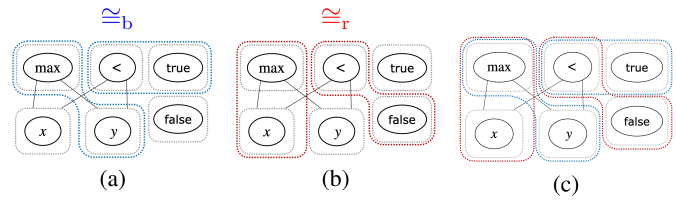


Fig. 1. Example e-graph with two colored layers; (a) is blue, (b) is red, (c) shows them combined.

II. Overview

From this point we assume familiarity with the basic e-graph structure which includes a union-find, hashcons, and an e-class map, as well as the basic operations of add, merge, rebuild, and e-matching (and consequently rewriting). For readers unfamiliar with e-graphs, or with deferred rebuilding, which was introduced in [30], additional background is given in Appendix A.

Colored E-graphs are an extension of e-graphs devised to add a generic approach for supporting conditional reasoning to e-graphs. Existing exploratory reasoning systems such as TheSy [26] and Ruler [20] utilize equality saturation with e-graphs for discovering new rewrite rules, but are limited in the presence of conditionals. For example, let $t := \mathsf{max}(x, y)$, then reasoning about the cases $x < y$ and $x \geq y$ separately is desirable: in the first case $t \cong x$, and in the second $t \cong y$. Without any assumptions, we can say neither and rewriting of t is blocked. The approach in [26] involves a prover that creates an e-graph *clone* for each case in case splitting, such as for $x < y$ and $x \geq y$. This process, however, incurs high runtime and memory costs. Non-relevant terms in the e-graph are unnecessarily duplicated, and rewrites are redundantly applied to these copies. Further case splits compound this issue, leading to an exponential increase in the number of clones with additional nested splits.

Colored e-graphs are designed to avoid duplication via sharing of the common terms, thus storing them only once when possible. The e-graph structure becomes *layered*: the lowermost layer represents a congruence relation over terms that is true in all cases (represented, normally, as e-classes containing e-nodes). On top of it are layered additional congruence relations that arise from various assumptions.

Going back to our example, the corresponding e-graph is shown in Figure 1, containing the terms $\mathsf{max}(x, y)$, $x < y$, true and false. Layers corresponding to assumptions $x < y$ and $x \geq y$ are shown in 1(a) and 1(b). To evoke intuition, we associate with each layer a unique *color*, and paint their e-classes (dotted outlines, in depicted e-graphs) accordingly. Conventionally, the lowermost layer is associated with the color black. In the subsequent example we will use blue for $x < y$ and red for $x \geq y$ when referring to the example. In the blue layer, $(x < y) \cong_{\mathrm{b}} \mathsf{true}$ and $\mathsf{max}(x, y) \cong_{\mathrm{b}} y$; in the red layer, $(x < y) \cong_{\mathrm{r}} \mathsf{false}$ and $\mathsf{max}(x, y) \cong_{\mathrm{r}} x$. This is shown via the corresponding blue and red dotted borders. Figure 1(c) shows a depiction where both colors are overlain on the same graph, which is a more faithful representation of the concept of colored e-graphs, although this visualization is clearly not scalable to larger graphs. In Figure 2, a larger graph can be seen that includes the terms $\mathsf{max}(x, y) - \mathsf{min}(x, y)$ and $|x - y|$. An overlain graph will be quite incomprehensible in this case, so the layers are shown separately; it can be easily discerned that $\mathsf{max}(x, y) - \mathsf{min}(x, y) \cong_{\mathrm{b}} |x - y|$ as well as $\mathsf{max}(x, y) - \mathsf{min}(x, y) \cong_{\mathrm{r}} |x - y|$.

Both additional layers, blue and red, use existing (black) e-nodes, with each color represented by further unions of e-classes in the black congruence relation. Each color's congruence $\cong_c$ is a *coarsening* of the black congruence, $\cong$, as $\cong\ \subseteq\ \cong_c$. In complex cases like the generalization of $\mathsf{max}(x, y) - \mathsf{min}(x, y) \cong |x - y|$ to $\mathsf{max}(x, y, z) - \mathsf{min}(x, y, z) \cong \mathsf{max}(|x - y|, |x - z|, |y - z|)$, the colored e-graphs have an important layered structure. This scenario requires reasoning about additional assumptions, building additional layers, such as $x < y \wedge y < z$ on top of $x < y$ (and respectively $x \geq y \wedge y < z$ on top of $x \geq y$). These additional layers will reuse the blue and red ones, as they are a coarsening of the respective $\cong_{\mathrm{b}}$ and $\cong_{\mathrm{r}}$.

Before diving into the design of colored e-graphs, it is better to start with their expected semantics. One way to understand the semantics of colored e-graphs is by analogy to a set of clones, i.e. separate e-graphs $\mathcal{E}$. One e-graph represents the base congruence $\cong$, and one e-graph per color c represents $\cong_c$. All e-graphs in $\mathcal{E}$ conceptually represent the same terms partitioned differently into e-classes. Thus, they have the same e-nodes, except that the choice of e-class id (the representative) may be different according to the composition of the e-classes. We will call the e-classes of the color congruences *colored e-classes*. A union in any layer, black or colored, is in effect a union applied to the respective e-graph and all its descendants. Thus, a union in the black layer (i.e. the original e-graph) is analogous to a union in *all* of the e-graphs of the corresponding e-classes; this maintains the invariant that every colored e-class is a union of (one or more) black e-classes. The colored e-graph semantics of the other operations—insertion, congruence closure, and e-matching—are the same as if they were performed across all clones.

A guiding observation in the design is that in equality saturation based exploratory reasoning tasks, where the e-graphs are extensive, each assumption leads to modest increase in congruences. Colored e-graphs are adapted to this scenario. The basic presupposition is that most colored layers, like the blue layer in Figure 2, do not involve an excessive amount of additional unions. In these cases, the space savings from not duplicating black e-nodes more than compensate for the added complexity in managing colored e-classes. With careful tweaks and a few optimizations, we show that we improve upon a clone-based approach. Importantly, if the assumption leads to an inordinate increase in additional unions, the clone-based approach could be more appropriate, and it is possible to use a clone for that specific assumption.

For presentation purposes, we start with a basic implementation that is not very efficient but is effective for understanding the concepts and data structures; then, we indicate some pain points, and move on to describe optimization steps that can alleviate them.

In the basic implementation, all e-nodes reside in the "black" layer, represented by a "vanilla" e-graph implemented in egg, with normal operations. The colored congruences do not have designated e-graphs of their own, and instead, the operations of merge, rebuild, and e-matching have *colored variants*, parameterized by an additional color c, that are semantically analogous to the same operations having been applied, in clone semantics, to the e-graph associated with color c in $\mathcal{E}$. (Insertion is deferred to later.)

Colored merge. In colored e-graphs, the union-find structure used for merging, which traditionally holds all e-class ids, is optimized. A master copy retains black unions, while each color layer has a *smaller* union-find for merged representative e-classes of the parent layer. This approach avoids replication of data across layers.

Colored e-matching. The e-class map is only saved for the black layer. This is sufficient, because an e-class in color c is always going to be a union of black e-classes, and all that is required for e-matching is finding e-nodes with a particular root (operator) in the course of the top-down traversal. So the union can be searched on demand by collecting all the "c-color siblings" of the e-class and searching them as well.

Colored congruence closure. In egg, the e-graph maintains congruence by cycling through a work list of altered classes, re-canonizing their parents, and identifying unions to complete congruence through duplicate detection. In colored e-graphs the root will behave the same, but for colored layers there is no single e-class, as the colored e-classes are a equality class of concrete e-classes. For each color, we maintain an additional work list and collect concrete parents from e-classes on demand. This results in a rebuild algorithm similar to egg's, but without updating the hashcons in colored layers, as they are not present.

For a more concrete example, we give a detailed walk-through of equality saturation in a colored e-graph of the red case from Figure 2(b), and show the steps taken to construct this colored layer in Appendix C.

When using the above operations in the context of equality saturation, e-matching is applied for all colors to discover matches for the left-hand sides of rules. For each match, the right-hand side of the rule needs to be inserted into the e-graph and merged or color-merged with the left-hand side. Inserting the e-nodes to the e-graphs makes them available to all layers. This aspect is sound, since we assume that the mere *existence* of a term in an e-graph does not in itself have the semantics of a judgement—it is only the placing e-nodes in the same e-class that asserts an equality. However, in the presence of many colors, and thus many colored matches, the result would be a large volume of e-nodes that are in black e-classes of size 1, as they were created to serve a single color. As

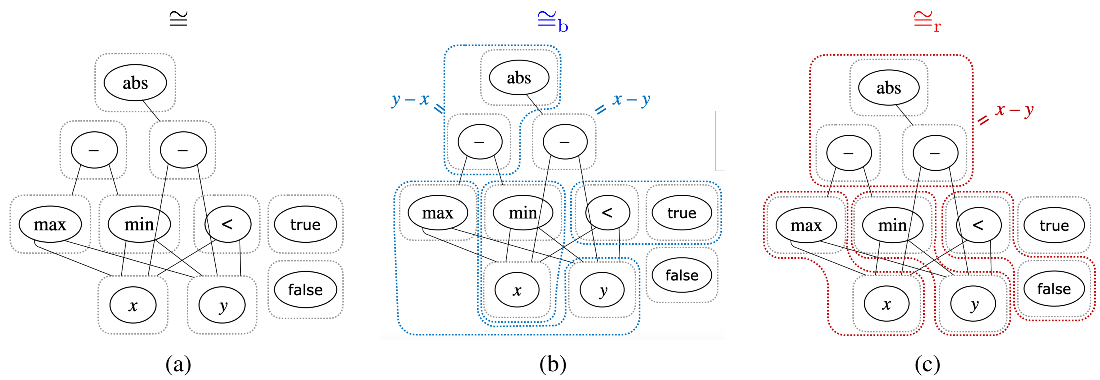


Fig. 2. Proof of $\max(x, y) - \min(x, y) = |x - y|$. The e-nodes corresponding to the two terms are in the same e-class both in the blue layer (b) and in the red (c). It is important to note that the layers are overlain, and that the black nodes are shared; they are separated here for ease of perception.

opposed to a, standard, single e-graph where merging e-classes shrinks the space of e-nodes (because non-equal e-nodes may become equal as a result of canonization), in colored unions it is required that the e-graph maintain both original e-classes, thus losing this advantage. This can put a growing pressure on subsequent e-matching and rebuild operations *in all colors*. Optimizations to improve colored e-graphs, and to address this issue, are presented in section IV.

III. Functional Description

We now introduce some notations and definitions that formalize the description of the e-graph presented in section II. We assume a term language L where terms are constructed using *function symbols*, each with its designated arity. We use $f^{(r)} \in \Sigma[L]$ to say that f is in the *signature* of L and has arity r. A term is then a *tree* whose nodes are labeled by function symbols and a node labeled by f has r children. (In particular, the leaves of a term have nullary function symbols.) Additionally we use the following definitions:

e-class ids	E
e-nodes	$N = \{f(e_1, .., e_r) \mid f^r \in \Sigma, e_i \in E\}$
union-find	$\equiv_{\text{id}} \subseteq E \times E$, $\equiv_{\text{id}}$ is an equivalence relation
e-class map	$M : E \to \mathcal{P}(N)$
parent map	$P = \{e \mapsto \{(n, e') \mid e' \in E \wedge n \in M(e') \wedge n = f(\ldots, e, \ldots)\} \mid e \in E\}$
hashcons	$H = \{n \mapsto e \mid n \in M(e)\}$

Semantically, every e-class represents a set of terms over Σ. We will use the notation $[t]$ to refer to e-class id of the equality class that represents (among other terms), the term t.

The union-find structure offers an operation, $\mathit{find}(e)$, that returns a unique representative id of the equivalence class (of $\equiv_{\text{id}}$) that contains e. That is, $\mathit{find}(e) \equiv_{\text{id}} e$ and for all $e_1 \equiv_{\text{id}} e_2$, $\mathit{find}(e_1) = \mathit{find}(e_2)$.

On top of these basic structures, we introduce a set of *colors*. As explained in section II, colors are organized in a tree whose root is the initial color ("black"). We mark the root color $\varnothing$ and assign to every non-root color c a *parent color* $p(c)$.

colors	$C = \{\varnothing, \ldots\}$
parent colors	$p : C \setminus \{\varnothing\} \to C$

The colored e-graph will now hold multiple union-find structures, one per color. They define a family of equivalence relations $\equiv_c$ by induction on the path from $\varnothing$ to c.

- ▷ $\equiv_\varnothing = \equiv_{\text{id}}$; $\mathit{find}_\varnothing(e) = \mathit{find}(e)$
- ▷ $\equiv_c \subseteq E_{p(c)} \times E_{p(c)}$, where $E_{p(c)} = \{\mathit{find}_{p(c)}(e) \mid e \in E\}$ is the set of all representatives from $\equiv_{p(c)}$. $\mathit{find}_c(e)$ for $e \in E_{p(c)}$ returns a unique identifier in the normal manner of union-find, *i.e.*, $\mathit{find}_c(e) \equiv_c e$ and for all $e_1 \equiv_c e_2$, $\mathit{find}_c(e_1) = \mathit{find}_c(e_2)$.

The definitions over $E_{p(c)}$ are naturally extended to E by (recursive) application of find; i.e., $\mathit{find}_c(e) = \mathit{find}_c(\mathit{find}_{p(c)}(e))$ and $e_1 \equiv_c e_2 \Leftrightarrow \mathit{find}_{p(c)}(e_1) \equiv_c \mathit{find}_{p(c)}(e_2)$. Thus it holds, by construction, that $\equiv_c \supseteq \equiv_{p(c)}$.

The colored e-graph also supports a $\mathit{merge}_c(e_1, e_2)$ operation for each color c where $e_1, e_2 \in E_c$. The merge operation may break the congruence relation invariants for c and all its descendants, and thus needs to be fixed. The merged classes are added to $\mathit{worklist}(c')$ for all c' where c' is c or one of its descendant. In egg [30], the invariants are restored periodically by performing a REBUILD pass. To accommodate the colors, we adjust the REBUILD logic to a multi-congruence-relation setting, so that it restores a congruence closure for each color during REBUILD. The main difference is that for a colored congruence relation, the procedure will collect the parents of a colored e-class by combining the sets of parents of all the (root) e-classes contained therein.

Another important colored e-graph operation is e-matching. Colored e-matching is a modification of the e-matching abstract machine presented in [19]. E-matching is performed by an abstract machine M which consists of a program counter, array of registers reg, and backtracking stack bs, in combination with a sequence of instructions that represents a pattern p. The machine will run instructions by order, where each may either fail if its assertion is not met, or produce a set of continuation states. If a continuation state is produced, the machine selects the first one and adds the current instruction to the stack. If no continuation state is produced, the machine backtracks, retrieving the most recent state from the stack and attempting the next available continuation.

To better present our modifications in colored egg, we first shortly introduce some of the original instruction types:

- ▷ $\mathsf{bind}(in, f, out)$ — Matches any e-node of the form $f(x_1, \ldots, x_n)$ that resides in the e-class saved in $reg[in]$, storing its children $x_{1..n}$ in $reg[out..out + n - 1]$.
- ▷ $\mathsf{compare}(i, j)$ — Asserts $reg[i] == reg[j]$.
- ▷ $\mathsf{check}(i, term)$ — Asserts that the e-class $reg[i]$ represents $term$.
- ▷ $\mathsf{continue}(f, out)$ — Match any e-node $f(x_1, \ldots, x_n)$ (in *any* e-class), storing its children $x_{1..n}$ in $reg[out..out + n - 1]$.
- ▷ $\mathsf{join}(in, reverse\_path, out)$ — Match any e-node $f(x_1, \ldots, x_n)$ that is reachable through $reverse\_path$ from the e-class $reg[in]$, storing its children $x_{1..n}$ in $reg[out..out + n - 1]$.

To facilitate matching across various congruence relations, we adjust the machine M to include the, currently being e-matched, colored assumption $color$ in its state. Adapting to $color$ involves changes in compilation and instructions. The two primary scenarios impacted are: during $\mathsf{compare}(i, j)$, ensuring $reg[i] \equiv_{color} reg[j]$, and in function application matching represented by a bind instruction. Before each 'bind' instruction, the modified compilation will insert a new 'colored_jump' instruction to try matching the full colored equality class, one "root" e-class at a time. This is achieved by having 'colored_jump(i)' yield all the "colored siblings" of $reg[i]$ in the current $color$, replacing $reg[i]$ with the result. The instruction 'check' can be likewise adjusted, but we point out that, in fact, it can be implemented as a sequence of 'bind's (with respective interleaved 'colored_jump's).

Multipatterns, supported by the abstract machine, enable e-matching against patterns with shared variables, useful for matching the precondition in conditional rewrite rules. This is achieved using the 'continue' instruction, which selects a new root for subsequent sub-patterns. In the colored setting, while 'continue' remains as is, for performance, it's sometimes substituted with 'join'. This alternative instruction also picks a new root, but restricts selection to e-nodes that can reach a specified e-class, linked to a previously matched hole, through child edges in the e-graph. A *reverse path* is provided to further restrict the upward search needed to find such e-nodes. We do not go too deep into the details, but its colored variant will invoke a colored_jump at every level. We point out that egg does not currently implement 'join', and our colored egg supports a special (though frequent) case in which $reverse\_path$ is empty.

The algorithms described here are presented in more depth in Appendix B.

IV. Optimizations

Both rebuilding and e-matching in colored e-graph, as discussed in section II, can be significantly slower compared to a separate, minimized e-graph.

In the rebuilding aspect, two main burdens are that the colored e-graph contains additional e-nodes compared to each of the separate ones, and that building a colored hash-cons (which will be presented shortly) requires going over all the e-classes.

In the e-matching aspect, colored e-matching may produce duplicate results due to the e-graph not being minimized according to the color's congruence relation; that is, colored-congruent terms are not always merged under a single e-class id. To illustrate this, consider a simple e-graph representing the terms $1 \cdot 1$, $1 \cdot x$, $1 \cdot y$, and $x \cdot y$. Introduce a color, blue, where $x \cong_{\mathrm{b}} y$. A simple pattern such as $1 \cdot ?v$ would have three matches, with assignments $?v \mapsto 1, ?v \mapsto x, ?v \mapsto y$. If the blue layer were a separate e-graph, x and y would have been in the same e-class, so one of the matches here is redundant (as far as the blue layer is concerned). Of course, in the black layer they are different matches; the point is, that many terms are added to the graph only as a result of a colored match, so matching them in the black e-graph is mostly useless to the reasoner. On the other hand, their *presence* in the black layer means they cannot ever be merged, leading to duplicate matches, as seen above, even in the respective colored layer(s).

Moreover, when inserting e-nodes to the e-graph, the hash-cons is used to prevent duplication, relying on it being canonized. Adding an e-node from a colored conclusion (following a match modulo $\cong_{\mathrm{b}}$) does not benefit from canonization. In fact, each e-node $f(x_1, \ldots, x_n)$ has a multitude of black representatives that are $\cong_{\mathrm{b}}$-equivalent. Each child x_i in the e-node can be presented by any black id such that $e \in [x_i]_b$, so there are $\prod_i |[x_i]_b|$ representations. These variants are distinct in the root color, so they cannot be de-duplicated as usual.

To address these issues, we present a series of optimizations to the colored e-graph data-structure and the procedures. These optimizations aim to reuse the "root" and ancestor layers as much as possible, both in terms of memory usage and compute. Thus, we can achieve a memory efficient, but effective colored e-graph.

A. Data-structure optimizations

Colored e-nodes. In the basic implementation outlined in section II, adding e-nodes from colored e-matches to the root e-graph may make it very large and increase the cost of all subsequent actions. The optimized version addresses this by introducing *colored e-nodes*, where e-nodes resulting from colored matches are tagged with their inducing colors. Each colored layer has its own colored hash-cons and e-class map, designed to store only the differences from the parent layer, thereby maximizing reuse. The new mappings added are:

e-class color	$EC : E \to C$
colored parent	$P_c = \{(n, e) \mid (n, e) \in P \wedge EC(e) = c\}$
colored hashcons	$H_c = \{n \mapsto e \mid n \in M(e) \wedge EC(e) = c\}$

Note that base parents and hashcons from the non-optimized version are incorporated as $P_\varnothing$ and $H_\varnothing$ in colored mappings.

This optimization applies the hierarchy in all operations. For example, while inserting an e-node to a color c, it is looked up in the colored hashcons for c and all its ancestors, $p^*(c)$, and finally, if no match is found, it is inserted into a new e-class e, setting $EC(e) = c$. The colored hashcons H_c is canonized to color c, ensuring that new e-nodes are unique to this layer and avoiding colored duplicates. (Some duplication related to c may still occur in ancestor layers, as their e-nodes are not canonized to c.) The optimization significantly impacts e-matching: previously when matching a function application f, all f-e-nodes in N were considered; now, only those e-nodes n in the colors hierarchy, that is, those satisfying $\exists e.\ n \in M(e) \wedge EC(e) \in p^*(c)$, are examined.

Pruning. Recall that having a coarsening relation between the colors in the hierarchy means that any result found in an ancestor color is also true for the descendant(s). And so, following merges, some of the colored e-nodes could become subsumed by e-nodes that already exist in an ancestor layer. We present an efficient deferred pruning method to remove the redundant e-nodes.

Normal e-graph minimization relies on having all e-nodes canonized. A colored e-graph usually does not canonize all e-nodes to a specific color c (except for $\varnothing$). Rather, H_c contains only the difference from previous layers. To find redundant e-nodes, the colored e-graph builds a transient hashcons during rebuild from all relevant e-nodes that are not c-colored. The new hashcons, H'_c, is created as follows:

$$H'_c = \{canonize_c(n) \mapsto find_c(e) \mid \\ n \in M(e), EC(e) \in p^+(c)\}$$

A c-colored class e can be reduced by removing all e-nodes that already exist in H'_c. While pruning is promising, one must take care that pruned e-nodes are not immediately re-added.

Colored minimization. Another improvement is having multiple colored e-nodes (of the same color) in a single (black) e-class. As mentioned previously, any e-node that resulted from a colored insert had to be in their own e-classes, as no black unions would be performed on them. But, given that $e \equiv_c e' \wedge EC(e) = EC(e') = c$, then the two black e-classes $e,\ e'$ can be merged as both contain colored e-nodes of the same color and are in the same colored e-class (of the same color). Thus an invariant is kept that each colored equality class has at most one black e-class containing colored e-nodes.

B. Procedure optimizations

Rebuild. When rebuilding, we first reconstruct the congruence relation of the "root" layer. Even though a color, for example blue, will need to rebuild its own congruence, it still holds that $\cong \subseteq \cong_b$. So, any union induced by $\cong$ can be applied to the blue relation. To understand the implications, consider the e-graph representing the terms x, y, $f(x)$, $f(y)$, $f(f(x))$, and $f(g(y))$ where the blue color contains the additional assumption that $g(y) \cong_b f(y)$. If we union x and y, the black congruence will include $f(x) \cong f(y)$ which also holds in the blue relation. But, the rebuilding of the blue congruence invariant will include an additional, deeper (in terms of rebuilding rounds), conclusion $f(f(x)) \cong_b f(g(y))$. This demonstrates how reusing parent relations is useful; the rebuild depth can be reduced by first rebuilding finer relations.

E-match. In e-matching, we implement an optimization where findings on the root layer are also valid for higher layers. To avoid redundant pattern matching, e-matching begins only from $\varnothing$, adding colored assumptions as needed. There are two scenarios for introducing a colored assumption: The first during *compare*(i, j), if $reg[i] \not\equiv_{color} reg[j]$, we explore descendant colors c where $reg[i] \equiv_c reg[j]$, adding states with $color \leftarrow c$ to the backtracking stack bs. The second is on-demand coloring in colored_jump, where jumps to any color c are enabled if $M.color \in p^+(c)$ and the target e-class is otherwise unreachable. We minimize the set of new assumptions to prevent redundant colors. During the updated compare, compare', if a color c is sufficient, its descendants are not added to bs. For to updated colored_jump, colored_jump', e-classes are matched only with their topmost (closest to root) congruent descendants. By taking the topmost descendants, we ensure that all additional matching paths are unique, as at least one (different) e-class is chosen at each fork. Despite eliminating duplicate paths, some duplicate colored matches persist due to incomplete minimization of the e-graph. The modified instructions are described in more detail in Appendix B.

V. Evaluation

Support for colored e-graphs is implemented in a modified version of egg, called Easter Egg. In this section, we evaluate the performance and effectiveness of Easter Egg and the different optimizations we presented. For this purpose we implemented two versions of colored e-graphs containing different improvements described in section IV. The simple version only uses procedural improvements, while the optimized version uses all optimizations.

A. Objectives and Evaluation Method

Our evaluation aims to test colored e-graphs' efficacy in equality saturation for exploratory reasoning tasks with multiple simultaneous assumptions. We evaluate the effectiveness using e-graph size and equality saturation time. To the best of our knowledge, a purely e-graph-based automated theorem prover does not exist, and theory exploration tools have limited support for conditions. Thus, for the evaluation, we created an equality saturation-based prover (based on code from [26]) that incorporates an automatic case-splitting mechanism.

The case-splitting mechanism is only used when it will potentially contribute to progress of the equality saturation process—that is, when it enables additional rewrite rules that were previously blocked. When this is detected, the prover yields appropriate assumptions, one for each case. We compare two settings: a baseline setting with separate e-graphs created by cloning, and Easter Egg's colored e-graph implementation.

We measure the total running times and the total size of all the e-graphs.

We evaluated our implementation on inductive proof suites from [24], also used in [26]. Since the instances are relatively small, we introduced a slight variation: for each goal, we combined benchmarks (i.e. proof goals) within the suite sharing similar goals and vocabulary. This approach generates larger benchmarks, and thus larger e-graphs, for more significant exploration, with the prover continuing until saturation or resource limit, regardless of early goal achievement. All the experiments were conducted on 64 core AMD EPYC 7742 processor with 512 GB RAM.

B. Experimental Setup

Using the enhanced prover, we evaluated each test case by measuring e-graph sizes and run times. E-graph size was determined by counting e-nodes; in colored layers, we tracked additional colored e-nodes, whereas for separate e-graphs, we measured the e-nodes in both the original and coarsened graphs. The experiments utilize the Cap library to cap memory usage at 32 GB and limit run-time to 1 hour per case.

Our experiments involved a basic colored e-graph implementation (as per section II which we dub monochrome colored e-graph, as it does not contain colored e-nodes) and a fully optimized version, comparing both against the baseline of separate e-graphs. The pruning optimization has almost identical results to the fully optimized version, and hence, for brevity, it is not shown. It is expected, due to pruning being ineffective in cases where the same rewrite rules are applied repeatedly, adding the removed e-nodes right back.

C. Results

In our setup, all assumptions emerge from case splits done by the prover. We filter out cases where no case splits were applied, since these have no assumptions introduced and thus colored e-graphs have no impact.

For each benchmark instance, we measure the *relative e-node overhead* as the number of additional e-nodes that are required, normalized by the number of different assumptions. That is, $(|\text{total e-nodes}|-|\text{base e-nodes}|)/|\text{assumptions}|$. "Base e-nodes" represent the contents of the graph before case splits. (For the monochrome colored e-graph we use the base e-nodes present in the separate e-graphs case.) Figure 3 summarizes the results, pitting colored e-graphs (with and without colored e-nodes) against the baseline of separate clones. In some cases one configuration times out or runs out of memory, while the other does not; we only compare cases where both configurations finished the run successfully. In both comparisons, we see roughly around $10\times$ lower overhead, where in the monochromatic case samples are more dispersed around the y axis, and the optimized case shows clear advantage to the colored e-graph implementation.

Run-time is measured as the the total run-time for completed test cases, and 1 hour for cases that timed out. We do not include runs that did not finish due to out-of-memory exceptions (we report the latter separately). As can be seen in Figure 4, the monochrome colored e-graph lead to many timeouts, whereas the optimized case exhibits running times similar to separate clones. This is in line with our expectation: colors provide lower memory sizes at the expense of run-time.

TABLE I
RUN-TIME AND EXCEPTIONS. M = OUT OF MEMORY, T = TIMEOUT (3600)

	Separate		Monochrome		Optimized	
Test Suite	Time	M/T	Time	M/T	Time	M/T
clam	70.1	0/0	277.8	0/5	23.6	0/0
hipspec-rev-equiv	34.1	0/0	139.0	0/17	57.0	0/0
hipspec-rotate	3880.3	1/1	1871.4	0/6	17.4	0/3
isaplanner	8454.4	0/60	6068.4	0/70	20486.3	3/28
leon-amortize-queue	187356.4	52/0	14.8	0/57	10854.3	3/49
leon-heap	1735.9	0/0	1201.8	0/25	4949.2	0/13

Finally, in Table I we present the number of out-of-memory exceptions, the number of timeout exceptions, and total run-time for each configurations and test suite. The monochrome colored e-graph, as expected, exhibits many timeouts. Even though it has more errors than the other e-graph versions, it still has much longer run-times.

The optimized e-graphs demonstrate enhancements over separate e-graphs in both run-time and success rate, as detailed in Table I. Notably, the optimized configuration completed more tests (99 failures compared to 114). A key shift observed is the replacement of out-of-memory errors with timeouts, particularly in the leon-amortize-queue suite. However, leon-heap posed challenges for colored e-graphs, incurring 13 extra timeouts even in the optimized version. Conversely, the isaplanner suite showed a notable improvement, halving the failure rate in the optimized version compared to the baseline.

VI. RELATED WORK

Theory exploration and its applications. Interest in exploratory reasoning in the context of functional calculi started with IsaCoSy [13], a system for lemma discovery based in part on CEGIS [28]. In a seminal paper, QuickSpec [27] propelled applicability of such reasoning for inferring specifications from implementations based on random testing, with deductive reasoning to verify generated conjectures [6], [12]. TheSy [26] and Ruler [20] have both incorporated e-graphs to some extent in the exploration process: they are used to speed up equivalence reduction of the space of generated terms, and, in [26], also the filtering and qualification phases using symbolic examples. The evaluation of the latter shows quite clearly that case splitting is a major obstacle to symbolic exploratory reasoning, due to the large number of different cases and derived assumptions.

In the area of conditional rewrite discovery, Speculate [4] naturally builds on the techniques from QuickSpec and depends on property-based testing techniques to generate inputs that satisfy some conditions. SWAPPER [25] is a relatively early example of exploring using SyGuS with a data-driven inductive-synthesis approach with emphasis on finding rules

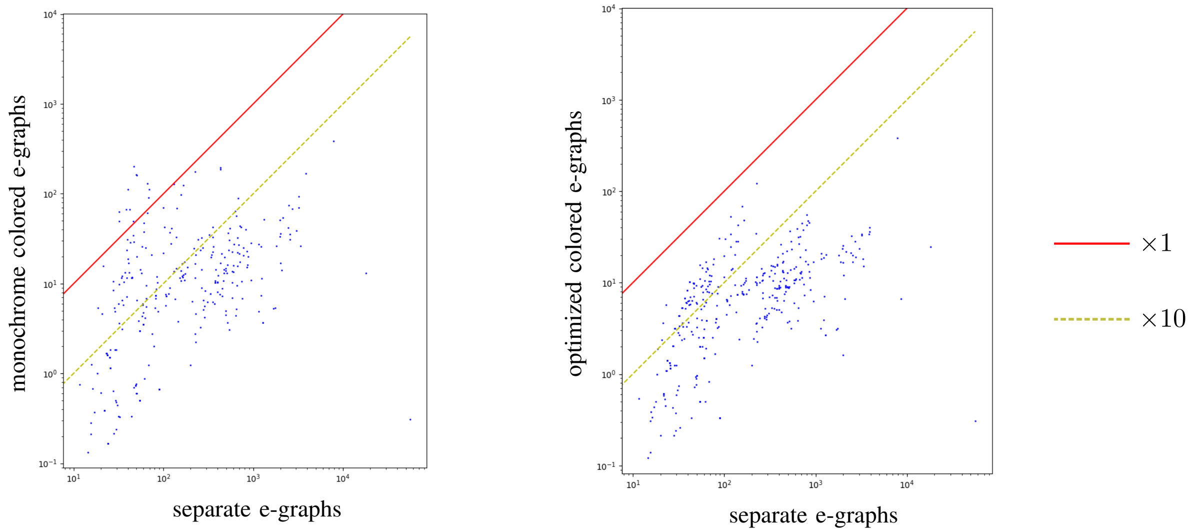


Fig. 3. Size comparison: relative e-node overhead in clones *vs.* color e-graph variants.

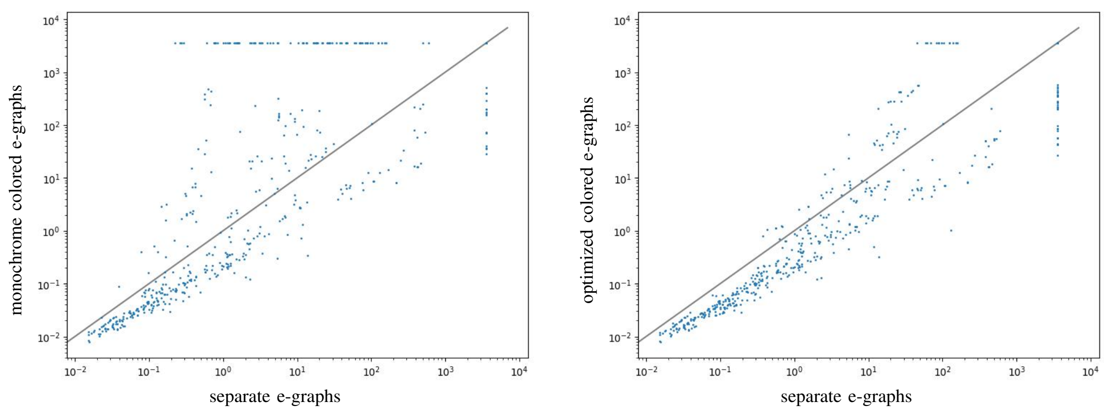


Fig. 4. Run-time comparison: run-time of clones vs. color e-graphs

that are most efficient for different problem domains. It requires a large corpus of similar SMT problems to operate.

Other e-graph extensions. E-graphs were originally brought into use for automated theorem proving [9], and were later popularized as a mechanism for implementing low-level compiler optimizations [29], by extending them with "φ-nodes" to express loops. Relational e-matching [32] makes use of Datalog seminaïve evaluation to harness the power of query planning in database systems. Subsequently, Datalog-powered e-matching has been recently fused with core Datalog semantics to allow richer logic programming by exposing equality saturation as a building block in a framework called egglog [31]. Since Datalog is based on Horn clauses, this meshes very well with conditional rewriting. It should be noted, though, that it is still a monotone framework, and does not allow backtracking or simultaneous exploration of alternative assumptions.

ECTAs [15], [11] are another, related compact data structure that extends e-graphs, Version-Space Algebras [17], [18], and Finite Tree Automata [1], with the concept of "entanglement"; that is, some choices of terms from e-classes may depend on choices done in other e-classes. Since the backbone of ECTAs is quite similar to an e-graph, the colors extension is applicable to this domain as well.

Uses of e-graphs in SMT. E-graphs are a core component for equality reasoning in SMT solvers [8], [2], in most theory solvers such as QF_UF, linear algebra, and bit-vectors. E-matching is also used for quantifier instantiation [21], which is, in its essence, an exploratory task and requires efficient methods [19]. In these contexts, implications and other Boolean structures are treated by the SAT core (in CDCL(T)), and the theory solver only handles conjunctions of literals.

VII. Conclusion

We presented colored e-graphs as an approach to efficiently handle multiple congruence relations in a single e-graph. They provide a memory-efficient method for equality saturation with additional assumptions, crucial for efficient exploratory reasoning of multiple assumptions simultaneously. Our optimizations, developed using the egg library, have shown notable improvements in memory usage and moderate enhancements in run-time performance over the baseline.

REFERENCES

[1] Adams, M.D., Might, M.: Restricting grammars with tree automata. Proc. ACM Program. Lang. **1**(OOPSLA), 82:1–82:25 (2017). https://doi.org/10.1145/3133906, https://doi.org/10.1145/3133906

[2] Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24, https://doi.org/10.1007/978-3-030-99524-9_24

[3] Bergstra, J., Klop, J.: Conditional rewrite rules: Confluence and termination. Journal of Computer and System Sciences **32**(3), 323–362 (1986). https://doi.org/https://doi.org/10.1016/0022-0000(86)90033-4, https://www.sciencedirect.com/science/article/pii/0022000086900334

[4] Braquehais, R., Runciman, C.: Speculate: discovering conditional equations and inequalities about black-box functions by reasoning from test results. In: Diatchki, I.S. (ed.) Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Oxford, United Kingdom, September 7-8, 2017. pp. 40–51. ACM (2017). https://doi.org/10.1145/3122955.3122961, https://doi.org/10.1145/3122955.3122961

[5] Brotherston, J., Gorogiannis, N., Petersen, R.L.: A generic cyclic theorem prover. In: Jhala, R., Igarashi, A. (eds.) Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7705, pp. 350–367. Springer (2012). https://doi.org/10.1007/978-3-642-35182-2_25, https://doi.org/10.1007/978-3-642-35182-2_25

[6] Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: Automating inductive proofs using theory exploration. In: International Conference on Automated Deduction. pp. 392–406. Springer (2013)

[7] Coward, S., Constantinides, G.A., Drane, T.: Automating constraint-aware datapath optimization using e-graphs. In: 2023 60th ACM/IEEE Design Automation Conference (DAC). pp. 1–6. IEEE (2023)

[8] De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)

[9] Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. J. ACM **52**(3), 365–473 (May 2005). https://doi.org/10.1145/1066100.1066102, https://doi.org/10.1145/1066100.1066102

[10] Flatt, O., Coward, S., Willsey, M., Tatlock, Z., Panchekha, P.: Small proofs from congruence closure. In: Griggio, A., Rungta, N. (eds.) 22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022. pp. 75–83. IEEE (2022). https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_13, https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_13

[11] Gissurarson, M.P., Roque, D., Koppel, J.: Spectacular: Finding laws from 25 trillion programs. In: ICST. vol. 6. Association for Computing Machinery, New York, NY, USA (2023)

[12] Johansson, M.: Automated theory exploration for interactive theorem proving: - an introduction to the hipster system. In: Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings. pp. 1–11 (2017). https://doi.org/10.1007/978-3-319-66107-0_1, https://doi.org/10.1007/978-3-319-66107-0_1

[13] Johansson, M., Dixon, L., Bundy, A.: Conjecture synthesis for inductive theories. Journal of Automated Reasoning **47**, 251–289 (2010)

[14] Jones, E., Ong, C.H.L., Ramsay, S.: Cycleq: an efficient basis for cyclic equational reasoning. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 395–409 (2022)

[15] Koppel, J., Guo, Z., de Vries, E., Solar-Lezama, A., Polikarpova, N.: Searching entangled program spaces. Proc. ACM Program. Lang. **6**(ICFP) (aug 2022). https://doi.org/10.1145/3547622, https://doi.org/10.1145/3547622

[16] Kovács, L., Voronkov, A.: First-order theorem proving and vampire. In: International Conference on Computer Aided Verification. pp. 1–35. Springer (2013)

[17] Lau, T.A., Domingos, P.M., Weld, D.S.: Version space algebra and its application to programming by demonstration. In: Langley, P. (ed.) Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), Stanford University, Stanford, CA, USA, June 29 - July 2, 2000. pp. 527–534. Morgan Kaufmann (2000)

[18] Lau, T.A., Wolfman, S.A., Domingos, P.M., Weld, D.S.: Programming by demonstration using version space algebra. Mach. Learn. **53**(1-2), 111–156 (2003). https://doi.org/10.1023/A:1025671410623, https://doi.org/10.1023/A:1025671410623

[19] de Moura, L.M., Bjørner, N.S.: Efficient e-matching for SMT solvers. In: Pfenning, F. (ed.) Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4603, pp. 183–198. Springer (2007). https://doi.org/10.1007/978-3-540-73595-3_13, https://doi.org/10.1007/978-3-540-73595-3_13

[20] Nandi, C., Willsey, M., Zhu, A., Wang, Y.R., Saiki, B., Anderson, A., Schulz, A., Grossman, D., Tatlock, Z.: Rewrite rule inference using equality saturation. Proc. ACM Program. Lang. **5**(OOPSLA), 1–28 (2021). https://doi.org/10.1145/3485496, https://doi.org/10.1145/3485496

[21] Niemetz, A., Preiner, M., Reynolds, A., Barrett, C.W., Tinelli, C.: Syntax-guided quantifier instantiation. In: Groote, J.F., Larsen, K.G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12652, pp. 145–163. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_8, https://doi.org/10.1007/978-3-030-72013-1_8

[22] Nötzli, A., Barbosa, H., Niemetz, A., Preiner, M., Reynolds, A., Barrett, C.W., Tinelli, C.: Reconstructing fine-grained proofs of rewrites using a domain-specific language. In: Griggio, A., Rungta, N. (eds.) 22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022. pp. 65–74. IEEE (2022). https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_12, https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_12

[23] Panchekha, P., Sanchez-Stern, A., Wilcox, J.R., Tatlock, Z.: Automatically improving accuracy for floating point expressions. ACM SIGPLAN Notices **50**(6), 1–11 (2015)

[24] Reynolds, A., Kuncak, V.: Induction for SMT solvers. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 80–98. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)

[25] Singh, R., Solar-Lezama, A.: SWAPPER: A framework for automatic generation of formula simplifiers based on conditional rewrite rules. In: Piskac, R., Talupur, M. (eds.) 2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016. pp. 185–192. IEEE (2016). https://doi.org/10.1109/FMCAD.2016.7886678, https://doi.org/10.1109/FMCAD.2016.7886678

[26] Singher, E., Itzhaky, S.: Theory exploration powered by deductive synthesis. In: Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II 33. pp. 125–148. Springer (2021)

[27] Smallbone, N., Johansson, M., Claessen, K., Algehed, M.: Quick specifications for the busy programmer. J. Funct. Program. **27**, e18 (2017). https://doi.org/10.1017/S0956796817000090, https://doi.org/10.1017/S0956796817000090

[28] Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006. pp. 404–415 (2006). https://doi.org/10.1145/1168857.1168907

[29] Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: A new approach to optimization. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 264–276. POPL '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1480881.1480915, https://doi.org/10.1145/1480881.1480915

[30] Willsey, M., Nandi, C., Wang, Y.R., Flatt, O., Tatlock, Z., Panchekha, P.: Egg: Fast and extensible equality saturation. Proc. ACM Program. Lang. **5**(POPL) (jan 2021). https://doi.org/10.1145/3434304, https://doi.org/10.1145/3434304

[31] Zhang, Y., Wang, Y.R., Flatt, O., Cao, D., Zucker, P., Rosenthal, E., Tatlock, Z., Willsey, M.: Better together: Unifying datalog and equality saturation. In: PLDI '23: 44rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (2023). https://doi.org/10.48550/arXiv.2304.04332, https://doi.org/10.48550/arXiv.2304.04332

[32] Zhang, Y., Wang, Y.R., Willsey, M., Tatlock, Z.: Relational e-matching. Proc. ACM Program. Lang. **6**(POPL), 1–22 (2022). https://doi.org/10.1145/3498696, https://doi.org/10.1145/3498696

APPENDIX A
BACKGROUND ON E-GRAPHS

We will now present some general background on e-graphs. Same as in section II, we assume a term language L where terms are constructed using *function symbols*, each with its designated arity. We use $f^{(r)} \in \Sigma[L]$ to say that f is in the *signature* of L and has arity r.

An e-graph $\mathcal{G}$ serves as a compact data structure representing a set $S \subseteq L$ of terms and a congruence relation $\cong \subseteq L \times L$. This congruence relation, in addition to being reflexive, symmetric, and transitive, is also closed under the function symbols of $\Sigma[L]$. That is, for every $f^r \in \Sigma[L]$, and given two lists of terms $t_{1..r} \in L$ and $s_{1..r}$, each of length r, if $t_i \cong s_i$ $(i = 1..r)$, then it follows that $f(t_1, \ldots, t_r) \cong f(s_1, \ldots, s_r)$. This property, known as *congruence closure*, is a key attribute of the data structure. The maintenance of this attribute as an invariant significantly influences the design and implementation of e-graph actions.

The egg library [30] revolutionizes the application of e-graphs by explicitly supporting the equality saturation workflow. It enables the periodic maintenance of congruence closure, via *deferred rebuild*, allowing for the amortization of associated rebuilding costs.

In egg, the authors present the e-graph as a union-find-like data structure, augmented to support operations on expressions. This implementation is primarily achieved through the utilization of three key structures: a hash-cons table, a union-find structure, and an e-class map. These structures collectively underpin the functionalities integral to the operation of the e-graph.

(a) The <u>union-find</u> component is responsible for keeping track of merged e-classes and maps each e-class id to a single representative for all (transitively) merged e-classes. This information is later used to canonicalize the keys and values of the hash-cons.
(b) The <u>e-class map</u> stores the structure of the e-graph. For each e-class id, the map keeps all the e-nodes that are contained therein. E-nodes are similar to AST nodes except that their children point to e-class ids instead of containing a single sub-term each.
(c) The <u>hash-cons</u> table maps e-nodes to their containing e-class id. An important aspect of the hash-cons is that after rebuilding, its keys and values are expected to be *canonical*. That is, whenever e-classes are merged one of their ids becomes "the" representative.

An e-class with id e represents a set of terms defined recursively as:

$$L(e) = \{f(t_1, .., t_k) \mid \\ f(e_1, .., e_k) \in M(e), t_i \in L(e_i) \text{ for } i = 1..k\}$$

We will use the notation $[t]$ to refer to e-class id where $t \in L([t])$.

Example A.1. The terms $\mathsf{max}(x, y)$ and $x - y$ are both represented in the e-graph in Figure 1(a) using e-classes $\langle 5\rangle$ and $\langle 6\rangle$, respectively, with the following e-nodes:

$$\begin{array}{lcll} M & = & \langle 1\rangle \mapsto \{\mathsf{true}\} & \langle 2\rangle \mapsto \{\mathsf{false}\} \\ & & \langle 3\rangle \mapsto \{x\} & \langle 4\rangle \mapsto \{y\} \\ & & \langle 5\rangle \mapsto \{\mathsf{max}(\langle 3\rangle, \langle 4\rangle)\} & \langle 6\rangle \mapsto \{\langle 3\rangle - \langle 4\rangle\} \end{array}$$

An e-graph where every e-class is a singleton, like this one, is just a forest of expression trees with sharing. The situation becomes more interesting once we start mutating the graph via its dedicated operations.

1) <u>Insert</u> - Adds a term t to the e-graph, one e-class per AST node, reusing e-classes where possible by searching the hash-cons.
2) <u>Merge</u> - Merging two e-classes by applying a union operation of the union-find and merging the classes in the e-class map. This, however, temporarily invalidates the invariant of the hash-cons and e-class map that all e-class ids and e-nodes must be canonical.
3) <u>Rebuilding (Congruence closure)</u> - As explained before, a union of $[x]$ into $[y]$ necessitates replacing any e-node $f([x], [z])$ by $f([y], [z])$. Moreover, if $f([x], [z]) \in [w_1], f([y], [z]) \in [w_2]$, then, following this replacement, both $[w_1]$ and $[w_2]$ now contain $f([y], [z])$, meaning that $[w_1] = [w_2]$ and evoking a cascading union of $[w_1], [w_2]$. A significant contribution by egg is the concept of deferred (and thus periodic) rebuilding. This periodic rebuilding is highly efficient and well-suited for equality saturation.
4) <u>E-matching</u> - Looking up a *pattern* in the set of terms represented by the e-graph in a top-down manner, traversing the e-nodes downward via the e-class map. A pattern is a term with (zero or more) *holes* represented by metavariables $?v_{1..k}$. For example, $(?v_1 + 1) \cdot ?v_2$ is a pattern. Pattern lookup is important for rewriting in equality saturation.

Rewriting. We assume a background set of symbolic *rewrite rules* (r.r.), each of the form $t \dot{\rightarrow} s$, where t and s are patterns as explained in item (4) above. A *match* θ of pattern t on the e-graph, is an assignment mapping metavariables to e-class ids. $t\theta$ represents an e-node, and we will denote its equality class as $[t\theta]$. Applying the r.r. is done by merging the e-classes $[s\theta]$ and $[t\theta]$. Because the e-node $s\theta$ might be new, it needs to also be inserted, resulting in $union([t\theta], insert(s\theta))$. Repetitively applying such rewrite rules to a set of terms can be used to generate growing sets of terms that are equivalent, according to rewrite semantics, to ones in the starting set. Ideally, the set eventually *saturates*, in which case the e-graph now describes *all* the terms that are rewrite-equivalent. We point out that in many situations, the e-graph keeps growing as a result of rewrites and never gets saturated—so the number of successive rewrite iterations, or "rewrite depth", has to be bounded.

A *conditional rewrite rule* (c.r.r.) [3] is a natural extension of a r.r. that has the following form: $\varphi \Rightarrow t \dot{\rightarrow} s$ where φ is a precondition for rewriting t to s. For example, the rules for max are: $?x > ?y \Rightarrow \mathsf{max}(?x, ?y) \dot{\rightarrow} ?x$ and $?x \leq ?y \Rightarrow \mathsf{max}(?x, ?y) \dot{\rightarrow} ?y$. The semantics of a precondition φ is defined such that a term matching the pattern of φ must be unified with Boolean true in order for the rewrite to be applied.

Appendix B
Algorithms Pseudo Code

Colored e-graphs introduce a few algorithmic changes to the operations of a normal e-graph. Here we present pseudo code for the important changes presented in the paper. Algorithm 1 presents the changes being made to the e-matching abstract machine to support *unoptimized* colored e-matching as presented in section III.

Algorithm 1 Instructions: compare and colored_jump

```
function COMPARE(i, j)
    if find(color, reg[i]) ≠ find(color, reg[j]) then
        backtrack
    end if
end function

function COLORED_JUMP(i)
    siblings ← {e | e ∈ E ∧ e ≡_color eclass}
    for sibling in siblings do
        reg[i] = sibling
        bs.push(current_state)
    end for
    backtrack
end function
```

The rebuilding algorithm is also updated to accommodate for colored e-graphs in section III, and the pseudo code in addition to some explanations is presented here. We update the auxiliary function REPAIR to work on colored e-classes, and introduce two new helper functions: COLLECT_PARENTS and UPDATE_HASHCONS, as presented in Algorithm 2. COLLECT_PARENTS extract the parents of a colored e-class by combining the sets of parents of all the (root) e-classes contained therein. UPDATE_HASHCONS is used to make sure that the hashcons entries are in canonical forms. It was already a part of REPAIR in egg; it is only repeated here to point out that it only updates the hashcons for the root color, since no canonization is required for colored layers.

The pseudo code for the optimized e-matching instructions that were presented in section IV are presented in Algorithm 4.

Algorithm 2 Colored Rebuilding

```
function REBUILD
    for color in self.colors do
        while self.worklist(color).len() > 0 do
            ▷ empty the worklist into a local variable
            todo ← TAKE(self.worklist(color))
            ▷ canonicalize and deduplicate the eclass refs to save calls to repair
            todo ← {self.find(color, eclass) | eclass ∈ todo}
            for each eclass in todo do
                SELF.REPAIR(color, eclass)
            end for
        end while
    end for
end function

function REPAIR(color, eclass)
    parents ← COLLECT_PARENTS(color, eclass)
    UPDATE_HASHCONS(color, parents)
    ▷ deduplicate the parents; note that equal parents get merged and put on the worklist
    new_parents ← {}
    for each (p_node, p_eclass) in parents do
        p_node ← self.canonicalize(color, p_node)
        if p_node is in new_parents then
            self.merge(color, p_eclass, new_parents[p_node])
            new_parents[p_node] ← self.find(color, p_eclass)
        end if
    end for
    if color = ∅ then
        eclass.parents ← new_parents
    end if
end function
```

The semantics of a conditional rewrite rule in the domain of an e-graph is that the condition pattern should be matched and its root must be in the same e-class as true, and, additionally, the left-hand side should be matched as normal. For simplicity of presentation, we pretend that $\neg$ is a special case were the negated condition is e-matched and the e-class should contain false.

Appendix C
Walkthrough for Example 2

This is the full walkthrough of the example in Figure 1 from the overview.

We walk through the steps needed to carry out the case splitting shown in Figure 2. The system contains the conditional rewrite rules shown on the right of Figure 5, which constitute the definitions of max and min, plus some prior knowledge about $|\cdot|$ and $-$.

Starting with the base graph, Figure 2(a), we describe the operation of Easter Egg on the red color, corresponding to the case $\neg x < y$. The complement blue case $(x < y)$ is analogous.

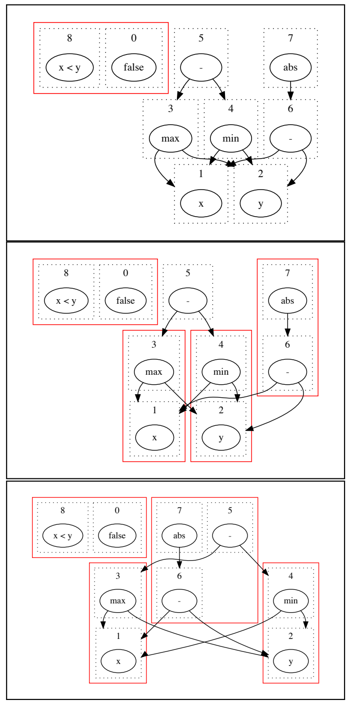
1) The value of $x < y$ is declared as false via a colored merge. This yields a new red e-class.
2) Colored e-matching is performed against the premise of the c.r.r. $\neg ?x < ?y \Rightarrow \mathsf{max}(?x, ?y) \dot{\rightarrow} ?x$. The condition of the rule, $?x < ?y$, matches against the class $[x < y]$, which is indeed in the same red e-class as false.
Similar e-matches are carried out for the rules $\neg ?x < ?y \Rightarrow \mathsf{min}(?x, ?y) \dot{\rightarrow} ?y$ and $\neg ?x < ?y \Rightarrow |?x - ?y| \dot{\rightarrow}$

(1)
merge$_{\color{red}{\mathrm{r}}}$([$x<y$], [false])

(2)
merge$_{\color{red}{\mathrm{r}}}$([max$(x,y)$], [$x$])
merge$_{\color{red}{\mathrm{r}}}$([min$(x,y)$], [$y$])
merge$_{\color{red}{\mathrm{r}}}$([$|x-y|$],
[$x-y$])

(3)
rebuild$_{\color{red}{\mathrm{r}}}$()
↓
merge$_{\color{red}{\mathrm{r}}}$([$x-y$],

max(x) − min(y))



rewrite rules

$?x<?y \Rightarrow \mathsf{min}(?x,?y) \dot{\to} ?x$
$\neg ?x<?y \Rightarrow \mathsf{min}(?x,?y) \dot{\to} ?y$
$?x<?y \Rightarrow \mathsf{max}(?x,?y) \dot{\to} ?y$
$\neg ?x<?y \Rightarrow \mathsf{max}(?x,?y) \dot{\to} ?x$
$?x<?y \Rightarrow |?x-?y| \dot{\to} ?y-?x$
$\neg ?x<?y \Rightarrow |?x-?y| \dot{\to} ?x-?y$

Fig. 5. Rewriting with case-split in a colored e-graph.

Algorithm 3 Colored Rebuilding (auxiliary methods)

```
function UPDATE_HASHCONS(color, parents)
    if color = ∅ then
        for each (p_node, p_eclass) in parents do
            self.hashcons.remove(p_node)
            p_node ← self.canonicalize(color, p_node)
            self.hashcons[p_node] ← self.find(color, p_eclass)
        end for
    end if
end function

function COLLECT_PARENTS(color, eclass)
    all_parents ← ∅                ▷ Initialize an empty set for parents
    relevant_eclasses ← {e | e ∈ E ∧ e ≡_color eclass}
    for e in relevant_eclasses do
        all_parents ← all_parents ∪ e.parents     ▷ Add parents of e to the set
    end for
    return all_parents
end function
```

$?x-?y$.

3) The children of $\langle 3\rangle - \langle 4\rangle$ ($\in M(\langle 5\rangle)$) are red-equivalent to those of $\langle 1\rangle - \langle 2\rangle$ ($\in M(\langle 6\rangle)$), and, as a consequence, red congruence closure kicks in and performs a red union there.

The process for blue is analogous. The case-split semantics is defined such that it records the fact that blue and red are *complements*, and as such extends $\equiv$ with the common equivalences, $\cong_{\mathrm{b}} \cap \cong_{\mathrm{r}} = \{\langle\langle 5\rangle, \langle 7\rangle\rangle, \ldots\}$.

Algorithm 4 Instructions: optimized compare and colored_jump

function COMPARE'(i, j)
  if $find(color, reg[i]) \neq find(color, reg[j])$ **then**
    $descendants \leftarrow \{c \mid color \in p^+(c) \wedge reg[i] \equiv_c reg[j]\}$
    $minimal \leftarrow \{c \mid c \in descendants \wedge \neg\exists c' \in descendants.\ c' \in p^+(c)\}$
    for c in $minimal$ **do**
      $color = c$
      bs.push(current_state)
    end for
    backtrack
  end if
end function

function COLORED_JUMP'(i)
  $siblings \leftarrow \{e \mid e \in E \wedge e \equiv_{color} eclass\}$
  for $sibling$ in $siblings$ **do**
    $reg[i] = sibling$
    bs.push(current_state)
  end for
  $descendants \leftarrow \{(c, e) \mid color \in p^+(c) \wedge reg[i] \equiv_c e \wedge e \notin siblings\}$
  $minimal \leftarrow \{(c, e) \mid (c, e) \in descendants \wedge \neg\exists(c', e') \in descendants.(c' \in p^+(c) \wedge e' \equiv'_c e)\}$
  for (c, e) in $minimal$ **do**
    $color = c$
    $reg[i] = e$
    bs.push(current_state)
  end for
  backtrack
end function