# egg: Fast and Extensible Equality Saturation

MAX WILLSEY, University of Washington, Seattle, USA
CHANDRAKANA NANDI, University of Washington, Seattle, USA
YISU REMY WANG, University of Washington, Seattle, USA
OLIVER FLATT, University of Utah, Salt Lake City, USA
ZACHARY TATLOCK, University of Washington, Seattle, USA
PAVEL PANCHEKHA, University of Utah, Salt Lake City, USA

An e-graph efficiently represents a congruence relation over many expressions. Although they were originally developed in the late 1970s for use in automated theorem provers, a more recent technique known as *equality saturation* repurposes e-graphs to implement state-of-the-art, rewrite-driven compiler optimizations and program synthesizers. However, e-graphs remain unspecialized for this newer use case. Equality saturation workloads exhibit distinct characteristics and often require ad hoc e-graph extensions to incorporate transformations beyond purely syntactic rewrites.

This work contributes two techniques that make e-graphs fast and extensible, specializing them to equality saturation. A new amortized invariant restoration technique called *rebuilding* takes advantage of equality saturation's distinct workload, providing asymptotic speedups over current techniques in practice. A general mechanism called *e-class analyses* integrates domain-specific analyses into the e-graph, reducing the need for ad hoc manipulation.

We implemented these techniques in a new open-source library called egg. Our case studies on three previously published applications of equality saturation highlight how egg's performance and flexibility enable state-of-the-art results across diverse domains.

## 1 INTRODUCTION

Equality graphs (e-graphs) were originally developed to efficiently represent congruence relations in automated theorem provers (ATPs). At a high level, e-graphs [Nelson 1980; Nieuwenhuis and Oliveras 2005] extend union-find [Tarjan 1975] to compactly represent equivalence classes of expressions while maintaining a key invariant: the equivalence relation is closed under congruence.[1]

Over the past decade, several projects have repurposed e-graphs to implement state-of-the-art, rewrite-driven compiler optimizations and program synthesizers using a technique known as *equality saturation* [Joshi et al. 2002; Nandi et al. 2020; Panchekha et al. 2015; Premtoon et al. 2020; Stepp et al. 2011; Tate et al. 2009; Wang et al. 2020]. Given an input program $p$, equality saturation constructs an e-graph $E$ that represents a large set of programs equivalent to $p$, and then extracts the "best" program from $E$. The e-graph is grown by repeatedly applying pattern-based rewrites. Critically, these rewrites only add information to the e-graph, eliminating the need for careful ordering. Upon reaching a fixed point (*saturation*), $E$ will represent *all equivalent ways* to express $p$ with respect to the given rewrites. After saturation (or timeout), a final *extraction* procedure analyzes $E$ and selects the optimal program according to a user-provided cost function.

Ideally, a user could simply provide a language grammar and rewrites, and equality saturation would produce a effective optimizer. Two challenges block this ideal. First, maintaining congruence can become expensive as $E$ grows. In part, this is because e-graphs from the conventional ATP setting remain unspecialized to the distinct *equality saturation workload*. Second, many applications critically depend on *domain-specific analyses*, but integrating them requires ad hoc extensions to the e-graph. The lack of a general extension mechanism has forced researchers to re-implement equality saturation from scratch several times [Panchekha et al. 2015; Tate et al. 2009; Wu et al. 2019]. These challenges limit equality saturation's practicality.

---

[1]Intuitively, congruence simply means that $a \equiv b$ implies $f(a) \equiv f(b)$.

*Equality Saturation Workload.*   ATPs frequently query and modify e-graphs and additionally require *backtracking* to undo modifications (e.g., in DPLL(T) [Davis and Putnam 1960]). These requirements force conventional e-graph designs to maintain the congruence invariant after every operation. In contrast, the equality saturation workload does not require backtracking and can be factored into distinct phases of (1) querying the e-graph to simultaneously find all rewrite matches and (2) modifying the e-graph to merge in equivalences for all matched terms.

We present a new amortized algorithm called *rebuilding* that defers e-graph invariant maintenance to equality saturation phase boundaries without compromising soundness. Empirically, rebuilding provides asymptotic speedups over conventional approaches.

*Domain-specific Analyses.*   Equality saturation is primarily driven by syntactic rewriting, but many applications require additional interpreted reasoning to bring domain knowledge into the e-graph. Past implementations have resorted to ad hoc e-graph manipulations to integrate what would otherwise be simple program analyses like constant folding.

To flexibly incorporate such reasoning, we introduce a new, general mechanism called *e-class analyses*. An e-class analysis annotates each e-class (an equivalence class of terms) with facts drawn from a semilattice domain. As the e-graph grows, facts are introduced, propagated, and joined to satisfy the *e-class analysis invariant*, which relates analysis facts to the terms represented in the e-graph. Rewrites cooperate with e-class analyses by depending on analysis facts and adding equivalences that in turn establish additional facts. Our case studies and examples (Sections 5 and 6) demonstrate e-class analyses like constant folding and free variable analysis which required bespoke customization in previous equality saturation implementations.

egg.   We implement rebuilding and e-class analyses in an open-source[2] library called egg (**e**-**g**raphs **g**ood). egg specifically targets equality saturation, taking advantage of its workload characteristics and supporting easy extension mechanisms to provide e-graphs specialized for program synthesis and optimization. egg also addresses more prosaic challenges, e.g., parameterizing over user-defined languages, rewrites, and cost functions while still providing an optimized implementation. Our case studies demonstrate how egg's features constitute a general, reusable e-graph library that can support equality saturation across diverse domains.

In summary, the contributions of this paper include:

- Rebuilding (Section 3), a technique that restores key correctness and performance invariants only at select points in the equality saturation algorithm. Our evaluation demonstrates that rebuilding is faster than existing techniques in practice.
- E-class analysis (Section 4), a technique for integrating domain-specific analyses that cannot be expressed as purely syntactic rewrites. The e-class analysis invariant provides the guarantees that enable cooperation between rewrites and analyses.
- A fast, extensible implementation of e-graphs in a library dubbed egg (Section 5).
- Case studies of real-world, published tools that use egg for deductive synthesis and program optimization across domains such as floating point accuracy, linear algebra optimization, and CAD program synthesis (Section 6). Where previous implementations existed, egg is orders of magnitude faster and offers more features.

## 2   BACKGROUND

egg builds on e-graphs and equality saturation. This section describes those techniques and presents the challenges that egg addresses.

---

[2]web: https://egraphs-good.github.io, source: https://github.com/egraphs-good/egg, documentation: https://docs.rs/egg

$$\begin{array}{rll}
\text{function symbols} & f, g & \\
\text{e-class ids} & a, b & \text{opaque identifiers} \\
\text{terms} & t ::= f \mid f(t_1, \ldots, t_m) & m \geq 1 \\
\text{e-nodes} & n ::= f \mid f(a_1, \ldots, a_m) & m \geq 1 \\
\text{e-classes} & c ::= \{n_1, \ldots, n_m\} & m \geq 1
\end{array}$$

Fig. 1. Syntax and metavariables for the components of an e-graph. Function symbols may stand alone as constant e-nodes and terms. An e-class id is an opaque identifier that can be compared for equality with =.

## 2.1 E-graphs

An *e-graph* is a data structure that stores a set of terms and a congruence relation over those terms. Originally developed for and still used in the heart of theorem provers [De Moura and Bjørner 2008; Detlefs et al. 2005; Nelson 1980], e-graphs have also been used to power a program optimization technique called *equality saturation* [Joshi et al. 2002; Nandi et al. 2020; Panchekha et al. 2015; Premtoon et al. 2020; Stepp et al. 2011; Tate et al. 2009; Wang et al. 2020].

*2.1.1 Definitions.* Intuitively, an e-graph is a set of equivalence classes (*e-classes*). Each e-class is a set of *e-nodes* representing equivalent terms from a given language, and an e-node is a function symbol paired with a list of children e-classes. More precisely:

*Definition 2.1 (Definition of an E-graph).* Given the definitions and syntax in Figure 1, an *e-graph* is a tuple $(U, M, H)$ where:

- A union-find data structure [Tarjan 1975] $U$ stores an equivalence relation (denoted with $\equiv_{\mathrm{id}}$) over e-class ids.
- The *e-class map* $M$ maps e-class ids to e-classes. All equivalent e-class ids map to the same e-class, i.e., $a \equiv_{\mathrm{id}} b$ iff $M[a]$ is the same set as $M[b]$. An e-class id $a$ is said to *refer to* the e-class $M[\mathrm{find}(a)]$.
- The *hashcons*[3] $H$ is a map from e-nodes to e-class ids.

Note that an e-class has an identity (its canonical e-class id), but an e-node does not.[4] We use e-class id $a$ and the e-class $M[\mathrm{find}(a)]$ synonymously when clear from the context.

*Definition 2.2 (Canonicalization).* An e-graph's union-find $U$ provides a find operation that canonicalizes e-class ids such that $\mathrm{find}(U, a) = \mathrm{find}(U, b)$ iff $a \equiv_{\mathrm{id}} b$. We omit the first argument of find where clear from context.

- An e-class id $a$ is canonical iff $\mathrm{find}(a) = a$.
- An e-node $n$ is canonical iff $n = \mathrm{canonicalize}(n)$, where $\mathrm{canonicalize}(f(a_1, a_2, \ldots)) = f(\mathrm{find}(a_1), \mathrm{find}(a_2), \ldots)$.

*Definition 2.3 (Representation of Terms).* An e-graph, e-class, or e-node is said to *represent* a term $t$ if $t$ can be "found" within it. Representation is defined recursively:

- An e-graph represents a term if any of its e-classes do.
- An e-class $c$ represents a term if any e-node $n \in c$ does.
- An e-node $f(a_1, a_2, \ldots)$ represents a term $f(t_1, t_2, \ldots)$ if they have the same function symbol $f$ and e-class $M[a_i]$ represents term $t_i$.

---

[3]We use the term *hashcons* to evoke the memoization technique, since both avoid creating new duplicates of existing objects.
[4]Our definition of an e-graph reflects egg's design and therefore differs with some other e-graph definitions and implementations. In particular, making e-classes but not e-nodes identifiable is unique to our definition.

(a) Initial e-graph contains $(a \times 2)/2$.

(b) After applying rewrite $x \times 2 \rightarrow x \ll 1$.

(c) After applying rewrite $(x \times y)/z \rightarrow x \times (y/z)$.

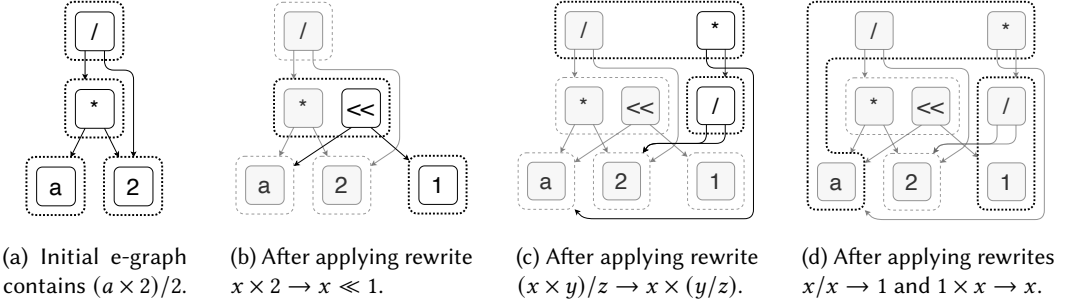(d) After applying rewrites $x/x \rightarrow 1$ and $1 \times x \rightarrow x$.

Fig. 2. An e-graph consists of e-classes (dashed boxes) containing equivalent e-nodes (solid boxes). Edges connect e-nodes to their child e-classes. Additions and modifications are emphasized in black. Applying rewrites to an e-graph adds new e-nodes and edges, but nothing is removed. Expressions added by rewrites are merged with the matched e-class. In Figure 2d, the rewrites do not add any new nodes, only merge e-classes. The resulting e-graph has a cycle, representing infinitely many expressions: $a$, $a \times 1$, $a \times 1 \times 1$, and so on.

When each e-class is a singleton (containing only one e-node), an e-graph is essentially a term graph with sharing. Figure 2a shows an e-graph that represents the expression $(a \times 2)/2$.

*Definition 2.4 (Equivalence).* An e-graph defines three equivalence relations.

- Over e-class ids: $a \equiv_{id} b$ iff $\texttt{find}(a) = \texttt{find}(b)$.
- Over e-nodes: $n_1 \equiv_{node} n_2$ iff e-nodes $n_1, n_2$ are in the same e-class, i.e., $\exists a.\ n_1, n_2 \in M[a]$.
- Over terms: $t_1 \equiv_{term} t_2$ iff terms $t_1, t_2$ are represented in the same e-class.

We use $\equiv$ without the subscript when the relation is clear from context.

*Definition 2.5 (Congruence).* For a given e-graph, let $\cong$ denote a congruence relation over e-nodes such that $f(a_1, a_2, ...) \cong f(b_1, b_2, ...)$ iff $a_i \equiv_{id} b_i$. Let $\cong^*$ denote the congruence closure of $\equiv_{node}$, i.e., the smallest superset of $\equiv_{node}$ that is also a superset of $\cong$. Note that there may be two e-nodes such that $n_1 \cong^* n_2$ but $n_1 \not\cong n_2$ and $n_1 \not\equiv_{node} n_2$. The relation $\cong$ only represents a single step of congruence; more than one step may be required to compute the congruence closure.

*2.1.2 E-graph Invariants.* The e-graph must maintain invariants in order to correctly and efficiently implement the operations given in Section 2.1.3. This section only defines the invariants, discussion of how they are maintained is deferred to Section 3. These are collectively referred to as the *e-graph invariants*.

*Definition 2.6 (The Congruence Invariant).* The equivalence relation over e-nodes must be closed over congruence, i.e., $(\equiv_{node}) = (\cong^*)$. The e-graph must ensure that congruent e-nodes are in the same e-class. Since identical e-nodes are trivially congruent, this implies that an e-node must be uniquely contained in a single e-class.

*Definition 2.7 (The Hashcons Invariant).* The hashcons $H$ must map all canonical e-nodes to their e-class ids. In other words:

$$\text{e-node } n \in M[a] \iff H[\texttt{canonicalize}(n)] = \texttt{find}(a)$$

If the hashcons invariant holds, then a procedure $\texttt{lookup}$ can quickly find which e-class (if any) has an e-node congruent to a given e-node $n$: $\texttt{lookup}(n) = H[\texttt{canonicalize}(n)]$.

*2.1.3 Interface and Rewriting.* E-graphs bear many similarities to the classic union-find data structure that they employ internally, and they inherit much of the terminology. E-graphs provide two main low-level mutating operations:

- add takes an e-node $n$ and:
  - if lookup($n$) = $a$, return $a$;
  - if lookup($n$) = $\emptyset$, then set $M[a] = \{n\}$ and return the id $a$.
- merge (sometimes called assert or union) takes two e-class ids $a$ and $b$, unions them in the union-find $U$, and combines the e-classes by setting both $M[a]$ and $M[b]$ to $M[a] \cup M[b]$.

Both of these operations must take additional steps to maintain the congruence invariant. Invariant maintenance is discussed in Section 3.

E-graphs also offers operations for querying the data structure.

- find canonicalizes e-class ids using the union-find $U$ as described in definition 2.1.
- ematch performs the *e-matching* [de Moura and Bjørner 2007; Detlefs et al. 2005] procedure for finding patterns in the e-graph. ematch takes a pattern term $p$ with variable placeholders and returns a list of tuples $(\sigma, c)$ where $\sigma$ is a substitution of variables to e-class ids such that $p[\sigma]$ is represented in e-class $c$.

These can be composed to perform rewriting over the e-graph. To apply a rewrite $\ell \rightarrow r$ to an e-graph, ematch finds tuples $(\sigma, c)$ where e-class $c$ represents $\ell[\sigma]$. Then, for each tuple, merge($c$, add($r[\sigma]$)) adds $r[\sigma]$ to the e-graph and unifies it with the matching e-class c.

Figure 2 shows an e-graph undergoing a series of rewrites. Note how the process is only additive; the initial term $(a \times 2)/2$ is still represented in the e-graph. Rewriting in an e-graph can also saturate, meaning the e-graph has learned every possible equivalence derivable from the given rewrites. If the user tried to apply $x \times y \rightarrow y \times x$ to an e-graph twice, the second time would add no additional e-nodes and perform no new merges; the e-graph can detect this and stop applying that rule.

## 2.2 Equality Saturation

Term rewriting [Dershowitz 1993] is a time-tested approach for equational reasoning in program optimization [Joshi et al. 2002; Tate et al. 2009], theorem proving [De Moura and Bjørner 2008; Detlefs et al. 2005], and program transformation [Andries et al. 1999]. In this setting, a tool repeatedly chooses one of a set of axiomatic rewrites, searches for matches of the left-hand pattern in the given expression, and replaces matching instances with the substituted right-hand side.

Term rewriting is typically destructive and "forgets" the matched left-hand side. Consider applying a simple strength reduction rewrite: $(a \times 2)/2 \rightarrow (a \ll 1)/2$. The new term carries no information about the initial term. Applying strength reduction at this point prevents us from canceling out 2/2. In the compilers community, this classically tricky question of when to apply which rewrite is called the *phase ordering* problem.

One solution to the phase ordering problem would simply apply all rewrites simultaneously, keeping track of every expression seen. This eliminates the problem of choosing the right rule, but a naive implementation would require space exponential in the number of given rewrites. *Equality saturation* [Stepp et al. 2011; Tate et al. 2009] is a technique to do this rewriting efficiently using an e-graph.

Figure 3 shows the equality saturation workflow. First, an initial e-graph is created from the input term. The core of the algorithm runs a set of rewrite rules until the e-graph is saturated (or a timeout is reached). Finally, a procedure called *extraction* selects the optimal represented term according to some cost function. For simple cost functions, a bottom-up, greedy traversal of the e-graph suffices to find the best term. Other extraction procedures have been explored for more complex cost functions [Wang et al. 2020; Wu et al. 2019].
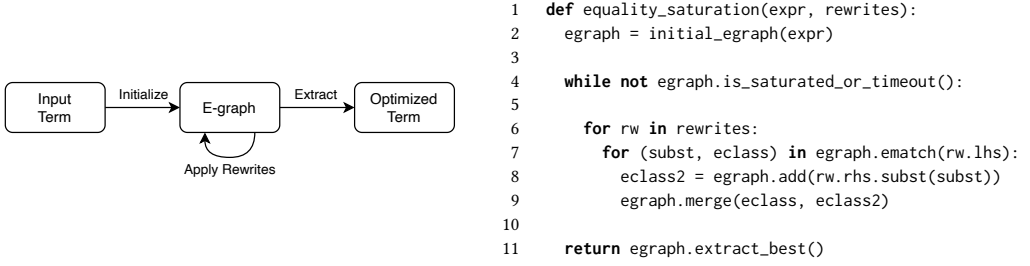
```
1   def equality_saturation(expr, rewrites):
2     egraph = initial_egraph(expr)
3
4     while not egraph.is_saturated_or_timeout():
5
6       for rw in rewrites:
7         for (subst, eclass) in egraph.ematch(rw.lhs):
8           eclass2 = egraph.add(rw.rhs.subst(subst))
9           egraph.merge(eclass, eclass2)
10
11    return egraph.extract_best()
```

Fig. 3. Box diagram and pseudocode for equality saturation. Traditionally, equality saturation maintains the e-graph data structure invariants throughout the algorithm.

Equality saturation eliminates the tedious and often error-prone task of choosing when to apply which rewrites, promising an appealingly simple workflow: state the relevant rewrites for the language, create an initial e-graph from a given expression, fire the rules until saturation, and finally extract the cheapest equivalent expression. Unfortunately, the technique remains ad hoc; prospective equality saturation users must implement their own e-graphs customized to their language, avoid performance pitfalls, and hack in the ability to do interpreted reasoning that is not supported by purely syntactic rewrites. egg aims to address each aspect of these difficulties.

## 2.3 Equality Saturation and Theorem Proving

An equality saturation engine and a theorem prover each have capabilities that would be impractical to replicate in the other. Automated theorem provers like satisfiability modulo theory (SMT) solvers are general tools that, in addition to supporting satisfiability queries, incorporate sophisticated, domain-specific solvers to allow interpreted reasoning within the supported theories. On the other hand, equality saturation is specialized for optimization, and its extraction procedure directly produces an optimal term with respect to a given cost function.

While SMT solvers are indeed the more general tool, equality saturation is not superseded by SMT; the specialized approach can be much faster when the full generality of SMT is not needed. To demonstrate this, we replicated a portion of the recent TASO paper [Jia et al. 2019], which optimizes deep learning models. As part of the work, they must verify a set of synthesized equalities with respect to a trusted set of universally quantified axioms. TASO uses Z3 [De Moura and Bjørner 2008] to perform the verification even though most of Z3's features (disjunctions, backtracking, theories, etc.) were not required. An equality saturation engine can also be used for verifying these equalities by adding the left and right sides of each equality to an e-graph, running the axioms as rewrites, and then checking if both sides end up in the same e-class. Z3 takes 24.65 seconds to perform the verification; egg performs the same task in 1.56 seconds (15× faster), or only 0.52 seconds (47× faster) when using egg's batched evaluation (Section 5.3).

## 3 REBUILDING: A NEW TAKE ON E-GRAPH INVARIANT MAINTENANCE

Traditionally [Detlefs et al. 2005; Nelson 1980], e-graphs maintain their data structure invariants after each operation. We separate this invariant restoration into a procedure called *rebuilding*. This separation allows the client to choose when to enforce the e-graph invariants. Performing a rebuild immediately after every operation replicates the traditional approach to invariant maintenance. In contrast, rebuilding less frequently can amortize the cost of invariant maintenance, significantly improving performance.

In this section, we first describe how e-graphs have traditionally maintained invariants (Section 3.1). We then describe the rebuilding framework and how it captures a spectrum of invariant maintenance approaches, including the traditional one (Section 3.2). Using this flexibility, we then give a modified algorithm for equality saturation that enforces the e-graph invariants at only select points (Section 3.3). We finally demonstrate that this new approach offers an asymptotic speedup over traditional equality saturation (Section 3.4).

## 3.1 Upward Merging

Both mutating operations on the e-graph (add and merge, Section 2.1.3) can break the e-graph invariants if not done carefully. E-graphs have traditionally used *hashconsing* and *upward merging* to maintain the congruence invariant.

The add operation relies on the hashcons invariant (Definition 2.7) to quickly check whether the e-node $n$ to be added—or one congruent to it—is already present. Without this check, add would create a new e-class with $n$ in it even if some $n' \cong n$ was already in the e-graph, violating the congruence invariant.

The merge operation e-classes can violate both e-graph invariants. If $f(a, b)$ and $f(a, c)$ reside in two different e-classes $x$ and $y$, merging $b$ and $c$ should also merge $x$ and $y$ to maintain the congruence invariant. This can propagate further, requiring additional merges.

E-graphs maintain a *parent list* for each e-class to maintain congruence. The parent list for e-class $c$ holds all e-nodes that have $c$ as a child. When merging two e-classes, e-graphs inspect these parent lists to find parents that are now congruent, recursively "upward merging" them if necessary.

The merge routine must also perform bookkeeping to preserve the hashcons invariant. In particular, merging two e-classes may change how parent e-nodes of those e-classes are canonicalized. The merge operation must therefore remove, re-canonicalize, and replace those e-nodes in the hashcons. In existing e-graph implementations [Panchekha et al. 2015] used for equality saturation, maintaining the invariants while merging can take the vast majority of run time.

## 3.2 Rebuilding in Detail

Traditionally, invariant restoration is part of the merge operation itself. Rebuilding separates these concerns, reducing merge's obligations and allowing for amortized invariant maintenance. In the rebuilding paradigm, merge maintains a *worklist* of e-class ids that need to be "upward merged", i.e., e-classes whose parents are possibly congruent but not yet in the same e-class. The rebuild operation processes this worklist, restoring the invariants of deduplication and congruence. Rebuilding is similar to other approaches in how it restores congruence (see Related Work for comparison to Downey et al. [1980]); but it uniquely allows the client to choose when to restore invariants in the context of a larger algorithm like equality saturation.

Figure 4 shows pseudocode for the main e-graph operations and rebuilding. Note that add and canonicalize are given for completeness, but they are unchanged from the traditional e-graph implementation. The merge operation is similar, but it only adds the new e-class to the worklist instead of immediately starting upward merging. Adding a call to rebuild right after the addition to the worklist (Figure 4 line 19) would yield the traditional behavior of restoring the invariants immediately.

The rebuild method essentially calls repair on the e-classes from the worklist until the worklist is empty. Instead of directly manipulating the worklist, egg's rebuild method first moves it into a local variable and deduplicates e-classes up to equivalence. Processing the worklist may merge e-classes, so breaking the worklist into chunks ensures that e-class ids made equivalent in the previous chunk are deduplicated in the subsequent chunk.

```
1   def add(enode):                              27   def rebuild():
2     enode = self.canonicalize(enode)          28     while self.worklist.len() > 0:
3     if enode in self.hashcons:                29       # empty the worklist into a local variable
4       return self.hashcons[enode]             30       todo = take(self.worklist)
5     else:                                     31       # canonicalize and deduplicate the eclass refs
6       eclass_id = self.new_singleton_eclass(enode)  32     # to save calls to repair
7       for child in enode.children:            33       todo = { self.find(eclass) for eclass in todo }
8         child.parents.add(enode, eclass_id)   34       for eclass in todo:
9       self.hashcons[enode] = eclass_id        35         self.repair(eclass)
10      return eclass_id                        36
11                                              37   def repair(eclass):
12  def merge(id1, id2)                         38     # update the hashcons so it always points
13    if self.find(id1) == self.find(id2):      39     # canonical enodes to canonical eclasses
14      return self.find(id1)                   40     for (p_node, p_eclass) in eclass.parents:
15    new_id = self.union_find.union(id1, id2)  41       self.hashcons.remove(p_node)
16    # traditional egraph merge can be         42       p_node = self.canonicalize(p_node)
17    # emulated by calling rebuild right after 43       self.hashcons[p_node] = self.find(p_eclass)
18    # adding the eclass to the worklist       44
19    self.worklist.add(new_id)                 45     # deduplicate the parents, noting that equal
20    return new_id                             46     # parents get merged and put on the worklist
21                                              47     new_parents = {}
22  def canonicalize(enode)                     48     for (p_node, p_eclass) in eclass.parents:
23    new_ch = [self.find(e) for e in enode.children]  49   p_node = self.canonicalize(p_node)
24    return mk_enode(enode.op, new_ch)         50       if p_node in new_parents:
25                                              51         self.merge(p_eclass, new_parents[p_node])
26  def find(eclass_id):                        52       new_parents[p_node] = self.find(p_eclass)
27    return self.union_find.find(eclass_id)    53     eclass.parents = new_parents
```

Fig. 4. Pseudocode for the add, merge, rebuild, and supporting methods. In each method, self refers to the e-graph being modified.

The actual work of rebuild occurs in the repair method. repair examines an e-class $c$ and first canonicalizes e-nodes in the hashcons that have $c$ as a child. Then it performs what is essentially one "layer" of upward merging: if any of the parent e-nodes have become congruent, then their e-classes are merged and the result is added to the worklist.

Deduplicating the worklist, and thus reducing calls to repair, is at the heart of why deferring rebuilding improves performance. Intuitively, the upward merging process of rebuilding traces out a "path" of congruence through the e-graph. When rebuilding happens immediately after merge (and therefore frequently), these paths can substantially overlap. By deferring rebuilding, the chunk-and-deduplicate approach can coalesce the overlapping parts of these paths, saving what would have been redundant work. In our modified equality saturation algorithm (Section 3.3), deferred rebuilding is responsible for a significant, asymptotic speedup (Section 3.4).

*3.2.1 Examples of Rebuilding.* Deferred rebuilding speeds up congruence maintenance by amortizing the work of maintaining the hashcons invariant. Consider the following terms in an e-graph: $f_1(x), ..., f_n(x), y_1, ..., y_n$. Let the workload be merge$(x, y_1), ...,$ merge$(x, y_n)$. Each merge may change the canonical representation of the $f_i(x)$s, so the traditional invariant maintenance strategy could require $O(n^2)$ hashcons updates. With deferred rebuilding the merges happen before the hashcons invariant is restored, requiring no more than $O(n)$ hashcons updates.

Deferred rebuilding can also reduce the number of calls to repair. Consider the following $w$ terms in an e-graph, each nested under $d$ function symbols:

$$f_1(f_2(\ldots f_d(x_1))), \quad \ldots, \quad f_1(f_2(\ldots f_d(x_w)))$$

Note that $w$ corresponds the width of this group of terms, and $d$ to the depth. Let the workload be $w - 1$ merges that merge all the $x$s together: for $i \in [2, w]$, $\mathtt{merge}(x_1, x_i)$.

In the traditional upward merging paradigm where $\mathtt{rebuild}$ is called after every $\mathtt{merge}$, each $\mathtt{merge}(x_i, x_j)$ will require $O(d)$ calls to $\mathtt{repair}$ to maintain congruence, one for each layer of $f_i$s. Over the whole workload, this requires $O(wd)$ calls to $\mathtt{repair}$.

With deferred rebuilding, however, the $w - 1$ merges can all take place before congruence must be restored. Suppose the $x$s are all merged into an e-class $c_x$ When $\mathtt{rebuild}$ finally is called, the only element in the deduplicated worklist is $c_x$. Calling $\mathtt{repair}$ on $c_x$ will merge the e-classes of the $f_d$ e-nodes into an e-class $c_{f_d}$, adding the e-classes that contained those e-nodes back to the worklist. When the worklist is again deduplicated, $c_{f_d}$ will be the only element, and the process repeats. Thus, the whole workload only incurs $O(d)$ calls to $\mathtt{repair}$, eliminating the factor corresponding to the width of this group of terms. Figure 8 shows that the number calls to $\mathtt{repair}$ is correlated with time spent doing congruence maintenance.

*3.2.2 Proof of Congruence.* Intuitively, rebuilding is a delay of the upward merging process, allowing the user to choose when to restore the e-graph invariants. They are substantially similar in structure, with a critical a difference in when the code is run. Below we offer a proof demonstrating that rebuilding restores the e-graph congruence invariant.

THEOREM 3.1. *Rebuilding restores congruence and terminates.*

PROOF. Since rebuilding only merges congruent nodes, the congruence closure $\cong^*$ is fixed even though $\equiv_{\mathrm{node}}$ changes. When $(\equiv_{\mathrm{node}}) = (\cong^*)$, congruence is restored. Note that both $\equiv_{\mathrm{node}}$ and $\cong^*$ are finite. We therefore show that rebuilding causes $\equiv_{\mathrm{node}}$ to approach $\cong^*$. We define the set of incongruent e-node pairs as $I = (\cong^*) \setminus (\equiv_{\mathrm{node}})$; in other words, $(n_1, n_2) \in I$ if $n_1 \cong^* n_2$ but $n_1 \not\equiv_{\mathrm{node}} n_2$.

Due to the additive nature of equality saturation, $\equiv_{\mathrm{node}}$ only increases and therefore $I$ is non-increasing. However, a call to $\mathtt{repair}$ inside the loop of $\mathtt{rebuild}$ does not necessarily shrink $I$. Some calls instead remove an element from the worklist but do not modify the e-graph at all.

Let the set $W$ be the worklist of e-classes to be processed by $\mathtt{repair}$; in Figure 4, $W$ corresponds to $\mathtt{self.worklist}$ plus the unprocessed portion of the $\mathtt{todo}$ local variable. We show that each call to $\mathtt{repair}$ decreases the tuple $(|I|, |W|)$ lexicographically until $(|I|, |W|) = (0, 0)$, and thus rebuilding terminates with $(\equiv_{\mathrm{node}}) = (\cong^*)$.

Given an e-class $c$ from $W$, $\mathtt{repair}$ examines $c$'s parents for congruent e-nodes that are not yet in the same e-class:

- If at least one pair of $c$'s parents are congruent, rebuilding merges each pair $(p_1, p_2)$, which adds to $W$ but makes $I$ smaller by definition.
- If no such congruent pairs are found, do nothing. Then, $|W|$ is decreased by 1 since $c$ came from the worklist and $\mathtt{repair}$ did not add anything back.

Since $(|I|, |W|)$ decreases lexicographically, $|W|$ eventually reaches 0, so $\mathtt{rebuild}$ terminates. Note that $W$ contains precisely those e-classes that need to be "upward merged" to check for congruent parents. So, when $W$ is empty, $\mathtt{rebuild}$ has effectively performed upward merging. By Nelson [1980, Chapter 7], $|I| = 0$. Therefore, when rebuilding terminates, congruence is restored. □

## 3.3 Rebuilding and Equality Saturation

Rebuilding offers the choice of when to enforce the e-graph invariants, potentially saving work if deferred thanks to the deduplication of the worklist. The client is responsible for rebuilding at a time that maximizes performance without limiting the application.

```
1   def equality_saturation(expr, rewrites):          1   def equality_saturation(expr, rewrites):
2     egraph = initial_egraph(expr)                   2     egraph = initial_egraph(expr)
3                                                      3
4     while not egraph.is_saturated_or_timeout():      4     while not egraph.is_saturated_or_timeout():
5                                                      5       matches = []
6                                                      6
7       # reading and writing is mixed                7       # read-only phase, invariants are preserved
8       for rw in rewrites:                           8       for rw in rewrites:
9         for (subst, eclass) in egraph.ematch(rw.lhs):  9         for (subst, eclass) in egraph.ematch(rw.lhs):
10                                                     10           matches.append((rw, subst, eclass))
11          # in traditional equality saturation,     11
12          # matches can be applied right away        12       # write-only phase, temporarily break invariants
13          # because invariants are always maintained 13       for (rw, subst, eclass) in matches:
14          eclass2 = egraph.add(rw.rhs.subst(subst))  14         eclass2 = egraph.add(rw.rhs.subst(subst))
15          egraph.merge(eclass, eclass2)              15         egraph.merge(eclass, eclass2)
16                                                     16
17          # restore the invariants after each merge  17       # restore the invariants once per iteration
18          egraph.rebuild()                           18       egraph.rebuild()
19                                                     19
20    return egraph.extract_best()                     20    return egraph.extract_best()
```

(a) Traditional equality saturation alternates be-        (b) egg splits equality saturation iterations into read
tween searching and applying rules, and the               and write phases. The e-graph invariants are not
e-graph maintains its invariants throughout.              constantly maintained, but restored only at the end
                                                          of each iteration by the rebuild method (Section 3).

Fig. 5. Pseudocode for traditional and egg's version of the equality saturation algorithm.

egg provides a modified equality saturation algorithm to take advantage of rebuilding. Figure 5 shows pseudocode for both traditional equality saturation and egg's variant, which exhibits two key differences:

(1) Each iteration is split into a read phase, which searches for all the rewrite matches, and a write phase that applies those matches.[5]
(2) Rebuilding occurs only once per iteration, at the end.

egg's separation of the read and write phases means that rewrites are truly unordered. In traditional equality saturation, later rewrites in the given rewrite list are favored in the sense that they can "see" the results of earlier rewrites in the same iteration. Therefore, the results depend on the order of the rewrite list if saturation is not reached (which is common on large rewrite lists or input expressions). egg's equality saturation algorithm is invariant to the order of the rewrite list.

Separating the read and write phases also allows egg to safely defer rebuilding. If rebuilding were deferred in the traditional equality saturation algorithm, rules later in the rewrite list would be searched against an e-graph with broken invariants. Since congruence may not hold, there may be missing equivalences, resulting in missing matches. These matches will be seen after the rebuild during the next iteration (if another iteration occurs), but the false reporting could impact metrics collection, rule scheduling,[6] or saturation detection.

---

[5]Although the original equality saturation paper [Tate et al. 2009] does not have separate reading and writing phases, some e-graph implementations (like the one inside Z3 [De Moura and Bjørner 2008]) do separate these phases as an implementation detail. Ours is the first algorithm to take advantage of this by deferring invariant maintenance.
[6]An optimization introduced in Figure 5.2 that relies on an accurate count of how many times a rewrite was matched.
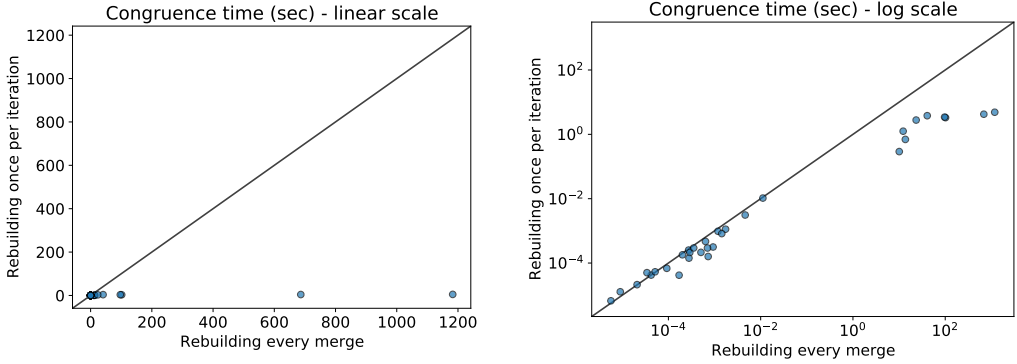
Fig. 6. Rebuilding once per iteration—as opposed to after every merge—significantly speeds up congruence maintenance. Both plots show the same data: one point for each of the 32 tests. The diagonal line is $y = x$; points below the line mean deferring rebuilding is faster. In aggregate over all tests (using geometric mean), congruence is 88× faster, and equality saturation is 21× faster. The linear scale plot shows that deferred rebuilding is significantly faster. The log scale plot suggests the speedup is greater than some constant multiple; Figure 7 demonstrates this in greater detail.
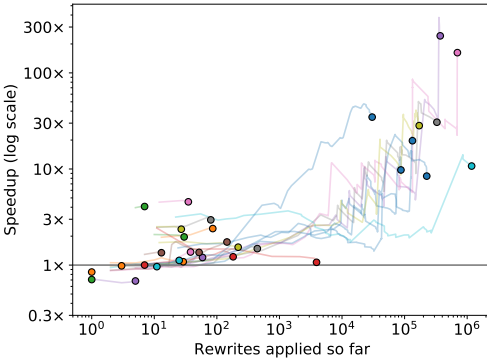


Fig. 7. As more rewrites are applied, deferring rebuilding gives greater speedup. Each line represents a single test: each equality saturation iteration plots the cumulative rewrites applied so far against the multiplicative speedup of deferring rebuilding; the dot represents the end of that test. Both the test suite as a whole (the dots) and individual tests (the lines) demonstrate an asymptotic speedup that increases with the problem size.
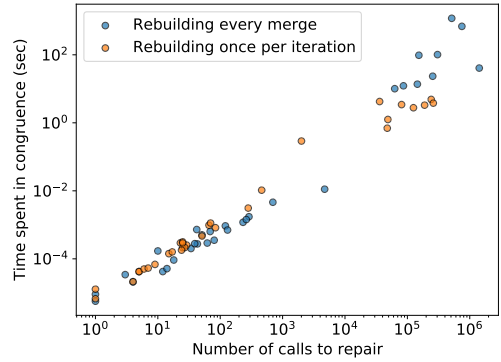
Fig. 8. The time spent in congruence maintenance correlates with the number of calls to the repair method. Spearman correlation yields $r = 0.98$ with a p-value of 3.6e-47, indicating that the two quantities are indeed positively correlated.

## 3.4 Evaluating Rebuilding

To demonstrate that deferred rebuilding provides faster congruence closure than traditional upward merging, we modified egg to call rebuild immediately after every merge. This provides a one-to-one comparison of deferred rebuilding against the traditional approach, isolated from the many other factors that make egg efficient: overall design and algorithmic differences, programming language performance, and other orthogonal performance improvements.

We ran egg's test suite using both rebuild strategies, measuring the time spent on congruence maintenance. Each test consists of one run of egg's equality saturation algorithm to optimize a given expression. Of the 32 total tests, 8 hit the iteration limit of 100 and the remainder saturated. Note that both rebuilding strategies use egg's phase-split equality saturation algorithm, and the resulting e-graphs are identical in all cases. These experiments were performed on a 2020 Macbook Pro with a 2 GHz quad-core Intel Core i5 processor and 16GB of memory.

Figure 6 shows our how rebuilding speeds up congruence maintenance. Overall, our experiments show an aggregate 88× speedup on congruence closure and 21× speedup over the entire equality saturation algorithm. Figure 7 shows this speedup is asymptotic; the multiplicative speedup increases as problem gets larger.

egg's test suite consists of two main applications: math, a small computer algebra system capable of symbolic differentiation and integration; and lambda, a partial evaluator for the untyped lambda calculus using explicit substitution to handle variable binding (shown in Section 5). Both are typical egg applications primarily driven by syntactic rewrites, with a few key uses of egg's more complex features like e-class analyses and dynamic/conditional rewrites.

egg can be configured to capture various metrics about equality saturation as it runs, including the time spent in the read phase (searching for matches), the write phase (applying matches), and rebuilding. In Figure 6, congruence time is measured as the time spent applying matches plus rebuilding. Other parts of the equality saturation algorithm (creating the initial e-graph, extracting the final term) take negligible take compared to the equality saturation iterations.

Deferred rebuilding amortizes the examination of e-classes for congruence maintenance; deduplicating the worklist reduces the number of calls to the repair. Figure 8 shows that time spent in congruence is correlated with the number of calls to the repair methods.

The case study in Section 6.1 provides a further evaluation of rebuilding. Rebuilding (and other egg features) have also been implemented in a Racket-based e-graph, demonstrating that rebuilding is a conceptual advance that need not be tied to the egg implementation.

## 4 EXTENDING E-GRAPHS WITH E-CLASS ANALYSES

As discussed so far, e-graphs and equality saturation provide an efficient way to implement a term rewriting system. Rebuilding enhances that efficiency, but the approach remains designed for purely syntactic rewrites. However, program analysis and optimization typically require more than just syntactic information. Instead, transformations are *computed* based on the input terms and also semantic facts about that input term, e.g., constant value, free variables, nullability, numerical sign, size in memory, and so on. The "purely syntactic" restriction has forced existing equality saturation applications [Panchekha et al. 2015; Stepp et al. 2011; Tate et al. 2009] to resort to ad hoc passes over the e-graph to implement analyses like constant folding. These ad hoc passes require manually manipulating the e-graph, the complexity of which could prevent the implementation of more sophisticated analyses.

We present a new technique called *e-class analysis*, which allows the concise expression of a program analysis over the e-graph. An e-class analysis resembles abstract interpretation lifted to the e-graph level, attaching *analysis data* from a semilattice to each e-class. The e-graph maintains and propagates this data as e-classes get merged and new e-nodes are added. Analysis data can be used directly to modify the e-graph, to inform how or if rewrites apply their right-hand sides, or to determine the cost of terms during the extraction process.

E-class analyses provide a general mechanism to replace what previously required ad hoc extensions that manually manipulate the e-graph. E-class analyses also fit within the equality saturation workflow, so they can naturally cooperate with the equational reasoning provided by rewrites. Moreover, an analysis lifted to the e-graph level automatically benefits from a sort of

"partial-order reduction" for free: large numbers of similar programs may be analyzed for little additional cost thanks to the e-graph's compact representation.

This section provides a conceptual explanation of e-class analyses as well as dynamic and conditional rewrites that can use the analysis data. The following sections will provide concrete examples: Section 5 discusses the egg implementation and a complete example of a partial evaluator for the lambda calculus; Section 6 discusses how three published projects have used egg and its unique features (like e-class analyses).

### 4.1 E-class Analyses

An e-class analysis defines a domain $D$ and associates a value $d_c \in D$ to each e-class $c$. The e-class $c$ contains the associated data $d_c$, i.e., given an e-class $c$, one can get $d_c$ easily, but not vice-versa.

The interface of an e-class analysis is as follows, where $G$ refers to the e-graph, and $n$ and $c$ refer to e-nodes and e-classes within $G$:

$\text{make}(n) \to d_c$     When a new e-node $n$ is added to $G$ into a new, singleton e-class $c$, construct a new value $d_c \in D$ to be associated with $n$'s new e-class, typically by accessing the associated data of $n$'s children.

$\text{join}(d_{c_1}, d_{c_2}) \to d_c$     When e-classes $c_1, c_2$ are being merged into $c$, join $d_{c_1}, d_{c_2}$ into a new value $d_c$ to be associated with the new e-class $c$.

$\text{modify}(c) \to c'$     Optionally modify the e-class $c$ based on $d_c$, typically by adding an e-node to $c$. Modify should be idempotent if no other changes occur to the e-class, i.e., $\text{modify}(\text{modify}(c)) = \text{modify}(c)$

The domain $D$ together with the join operation should form a join-semilattice. The semilattice perspective is useful for defining the *analysis invariant* (where $\wedge$ is the join operation):

$$\forall c \in G. \quad d_c = \bigwedge_{n \in c} \text{make}(n) \quad \text{and} \quad \text{modify}(c) = c$$

The first part of the analysis invariant states that the data associated with each e-class must be the join of the make for every e-node in that e-class. Since $D$ is a join-semilattice, this means that $\forall c, \forall n \in c, d_c \geq \text{make}(n)$. The motivation for the second part is more subtle. Since the analysis can modify an e-class through the modify method, the analysis invariant asserts that these modifications are driven to a fixed point. When the analysis invariant holds, a client looking at the analysis data can be assured that the analysis is "stable" in the sense that recomputing make, join, and modify will not modify the e-graph or any analysis data.

*4.1.1 Maintaining the Analysis Invariant.* We extend the rebuilding procedure from Section 3 to restore the analysis invariant as well as the congruence invariant. Figure 9 shows the necessary modifications to the rebuilding code from Figure 4.

Adding e-nodes and merging e-classes risk breaking the analysis invariant in different ways. Adding e-nodes is the simpler case; lines 10–11 restore the invariant for the newly created, singleton e-class that holds the new e-node. When merging e-nodes, the first concern is maintaining the semilattice portion of the analysis invariant. Since join forms a semilattice over the domain $D$ of the analysis data, the order in which the joins occur does not matter. Therefore, line 18 suffices to update the analysis data of the merged e-class.

Since make($n$) creates analysis data by looking at the data of $n$'s, children, merging e-classes can violate the analysis invariant in the same way it can violate the congruence invariant. The solution is to use the same worklist mechanism introduced in Section 3. Lines 37–44 of the repair method (which rebuild on each element of the worklist) re-make and merge the analysis data

```
 1  def add(enode):
 2    enode = self.canonicalize(enode)
 3    if enode in self.hashcons:
 4      return self.hashcons[enode]
 5    else:
 6      eclass = self.new_singleton_eclass(enode)
 7      for child_eclass in enode.children:
 8        child_eclass.parents.add(enode, eclass)
 9      self.hashcons[enode] = eclass
10      eclass.data = analysis.make(enode)
11      analysis.modify(eclass)
12      return eclass
13
14  def merge(eclass1, eclass2):
15    union = self.union_find.union(eclass1, eclass2)
16    if not union.was_already_unioned:
17      d1, d2 = eclass1.data, eclass2.data
18      union.eclass.data = analysis.join(d1, d2)
19      self.worklist.add(union.eclass)
20    return union.eclass
```

```
21  def repair(eclass):
22    for (p_node, p_eclass) in eclass.parents:
23      self.hashcons.remove(p_node)
24      p_node = self.canonicalize(p_node)
25      self.hashcons[p_node] = self.find(p_eclass)
26
27    new_parents = {}
28    for (p_node, p_eclass) in eclass.parents:
29      p_node = self.canonicalize(p_node)
30      if p_node in new_parents:
31        self.union(p_eclass, new_parents[p_node])
32      new_parents[p_node] = self.find(p_eclass)
33    eclass.parents = new_parents
34
35    # any mutations modify makes to eclass
36    # will add to the worklist
37    analysis.modify(eclass)
38    for (p_node, p_eclass) in eclass.parents:
39      new_data = analysis.join(
40        p_eclass.data,
41        analysis.make(p_node))
42      if new_data != p_eclass.data:
43        p_eclass.data = new_data
44        self.worklist.add(p_eclass)
```

Fig. 9. The pseudocode for maintaining the e-class analysis invariant is largely similar to how rebuilding maintains congruence closure (Section 3). Only lines 10–11, 17–18, and 37–44 are added. Grayed out or missing code is unchanged from Figure 4.

of the parent of any recently merged e-classes. The new repair method also calls modify once, which suffices due to its idempotence. In the pseudocode, modify is reframed as a mutating method for clarity.

egg's implementation of e-class analyses assumes that the analysis domain $D$ is indeed a semilattice and that modify is idempotent. Without these properties, egg may fail to restore the analysis invariant on rebuild, or it may not terminate.

*4.1.2 Example: Constant Folding.* The data produced by e-class analyses can be usefully consumed by other components of an equality saturation system (see Section 4.2), but e-class analyses can be useful on their own thanks to the modify hook. Typical modify hooks will either do nothing, check some invariant about the e-classes being merged, or add an e-node to that e-class (using the regular add and merge methods of the e-graph).

As mentioned above, other equality saturation implementations have implemented constant folding as custom, ad hoc passes over the e-graph. We can formulate constant folding as an e-class analysis that highlights the parallels with abstract interpretation. Let the domain $D =$ Option<Constant>, and let the join operation be the "or" operation of the Option type:

```
match (a, b) {
  (None,    None   ) => None,
  (Some(x), None   ) => Some(x),
  (None,    Some(y)) => Some(y),
  (Some(x), Some(y)) => { assert!(x == y); Some(x) }
}
```

Note how join can also aid in debugging by checking properties about values that are unified in the e-graph; in this case we assert that all terms represented in an e-class should have the same

constant value. The make operation serves as the abstraction function, returning the constant value of an e-node if it can be computed from the constant values associated with its children e-classes. The modify operation serves as a concretization function in this setting. If $d_c$ is a constant value, then modify$(c)$ would add $\gamma(d_c) = n$ to $c$, where $\gamma$ concretizes the constant value into a childless e-node.

Constant folding is an admittedly simple analysis, but one that did not formerly fit within the equality saturation framework. E-class analyses support more complicated analyses in a general way, as discussed in later sections on the egg implementation and case studies (Sections 5 and 6).

## 4.2 Conditional and Dynamic Rewrites

In equality saturation applications, most of the rewrites are purely syntactic. In some cases, additional data may be needed to determine if or how to perform the rewrite. For example, the rewrite $x/x \rightarrow 1$ is only valid if $x \neq 0$. A more complex rewrite may need to compute the right-hand side dynamically based on an analysis fact from the left-hand side.

The right-hand side of a rewrite can be generalized to a function apply that takes a substitution and an e-class generated from e-matching the left-hand side, and produces a term to be added to the e-graph and unified with the matched e-class. For a purely syntactic rewrite, the apply function need not inspect the matched e-class in any way; it would simply apply the substitution to the right-hand pattern to produce a new term.

E-class analyses greatly increase the utility of this generalized form of rewriting. The apply function can look at the analysis data for the matched e-class or any of the e-classes in the substitution to determine if or how to construct the right-hand side term. These kinds of rewrites can broken down further into two categories:

- *Conditional* rewrites like $x/x \rightarrow 1$ that are purely syntactic but whose validity depends on checking some analysis data;
- *Dynamic* rewrites that compute the right-hand side based on analysis data.

Conditional rewrites are a subset of the more general dynamic rewrites. Our egg implementation supports both. The example in Section 5 and case studies in Section 6 heavily use generalized rewrites, as it is typically the most convenient way to incorporate domain knowledge into the equality saturation framework.

## 4.3 Extraction

Equality saturation typically ends with an extraction phase that selects an optimal represented term from an e-class according to some cost function. In many domains [Nandi et al. 2020; Panchekha et al. 2015], AST size (sometimes weighted differently for different operators) suffices as a simple, local cost function. We say a cost function $k$ is local when the cost of a term $f(a_1, ...)$ can be computed from the function symbol $f$ and the costs of the children. With such cost functions, extracting an optimal term can be efficiently done with a fixed-point traversal over the e-graph that selects the minimum cost e-node from each e-class [Panchekha et al. 2015].

Extraction can be formulated as an e-class analysis when the cost function is local. The analysis data is a tuple $(n, k(n))$ where $n$ is the cheapest e-node in that e-class and $k(n)$ its cost. The make$(n)$ operation calculates the cost $k(n)$ based on the analysis data (which contain the minimum costs) of $n$'s children. The merge operation simply takes the tuple with lower cost. The semilattice portion of the analysis invariant then guarantees that the analysis data will contain the lowest-cost e-node in each class. Extract can then proceed recursively; if the analysis data for e-class $c$ gives $f(c_1, c_2, ...)$ as the optimal e-node, the optimal term represented in $c$ is extract$(c) = f(\text{extract}(c_1), \text{extract}(c_2), ...)$. This not only further demonstrates the generality of e-class analyses, but also provides the ability

to do extraction "on the fly"; conditional and dynamic rewrites can determine their behavior based on the cheapest term in an e-class.

Extraction (whether done as a separate pass or an e-class analysis) can also benefit from the analysis data. Typically, a local cost function can only look at the function symbol of the e-node $n$ and the costs of $n$'s children. When an e-class analysis is attached to the e-graph, however, a cost function may observe the data associated with $n$'s e-class, as well as the data associated with $n$'s children. This allows a cost function to depend on computed facts rather that just purely syntactic information. In other words, the cost of an operator may differ based on its inputs. Section 6.2 provides a motivating case study wherein an e-class analysis computes the size and shape of tensors, and this size information informs the cost function.

## 5 egg: EASY, EXTENSIBLE, AND EFFICIENT E-GRAPHS

We implemented the techniques of rebuilding and e-class analysis in egg, an easy-to-use, extensible, and efficient e-graph library. To the best of our knowledge, egg is the first general-purpose, reusable e-graph implementation. This has allowed focused effort on ease of use and optimization, knowing that any benefits will be seen across use cases as opposed to a single, ad hoc instance.

This section details egg's implementation and some of the various optimizations and tools it provides to the user. We use an extended example of a partial evaluator for the lambda calculus[7], for which we provide the complete source code (which few changes for readability) in Figure 10 and Figure 11. While contrived, this example is compact and familiar, and it highlights (1) how egg is used and (2) some of its novel features like e-class analyses and dynamic rewrites. It demonstrates how egg can tackle binding, a perennially tough problem for e-graphs, with a simple explicit substitution approach powered by egg's extensibility. Section 6 goes further, providing real-world case studies of published projects that have depended on egg.

egg is implemented in ~5000 lines of Rust,[8] including code, tests, and documentation. egg is open-source, well-documented, and distributed via Rust's package management system.[9] All of egg's components are generic over the user-provided language, analysis, and cost functions.

### 5.1 Ease of Use

egg's ease of use comes primarily from its design as a library. By defining only a language and some rewrite rules, a user can quickly start developing a synthesis or optimization tool. Using egg as a Rust library, the user defines the language using the define_language! macro shown in Figure 10, lines 1-22. Childless variants in the language may contain data of user-defined types, and e-class analyses or dynamic rewrites may inspect this data.

The user provides rewrites as shown in Figure 10, lines 51-100. Each rewrite has a name, a left-hand side, and a right-hand side. For purely syntactic rewrites, the right-hand is simply a pattern. More complex rewrites can incorporate conditions or even dynamic right-hand sides, both explained in the Section 5.2 and Figure 11.

Equality saturation workflows, regardless of the application domain, typically have a similar structure: add expressions to an empty e-graph, run rewrites until saturation or timeout, and extract the best equivalent expressions according to some cost function. This "outer loop" of equality saturation involves a significant amount of error-prone boilerplate:

---

[7]E-graphs do not have any "built-in" support for binding; for example, equality modulo alpha renaming is not free. The explicit substitution provided in this section is is illustrative but rather high in performance cost. Better support for languages with binding is important future work.

[8]Rust is a high-level systems programming language. egg has been integrated into applications written in other programming languages using both C FFI and serialization approaches.

[9]Source: https://github.com/mwillsey/egg. Documentation: https://docs.rs/egg. Package: https://crates.io/crates/egg.

```
1  define_language! {
2    enum Lambda {
3      // enum variants have data or children (eclass Ids)
4      // [Id; N] is an array of N `Id`s
5
6      // base type operators
7      "+" = Add([Id; 2]), "=" = Eq([Id; 2]),
8      "if" = If([Id; 3]),
9
10     // functions and binding
11     "app" = App([Id; 2]), "lam" = Lambda([Id; 2]),
12     "let" = Let([Id; 3]), "fix" = Fix([Id; 2]),
13
14     // (var x) is a use of `x` as an expression
15     "var" = Use(Id),
16     // (subst a x b) substitutes a for (var x) in b
17     "subst" = Subst([Id; 3]),
18
19     // base types have no children, only data
20     Bool(bool), Num(i32), Symbol(String),
21   }
22 }
23
24 // example terms and what they simplify to
25 // pulled directly from the egg test suite
26
27 test_fn! { lambda_under, rules(),
28   "(lam x (+ 4 (app (lam y (var y)) 4)))"
29   => "(lam x 8))",
30 }
31
32 test_fn! { lambda_compose_many, rules(),
33   "(let compose (lam f (lam g (lam x
34                 (app (var f)
35                   (app (var g) (var x))))))
36    (let add1 (lam y (+ (var y) 1))
37    (app (app (var compose) (var add1))
38        (app (app (var compose) (var add1))
39            (app (app (var compose) (var add1))
40                (app (app (var compose) (var add1))
41                    (var add1)))))))"
42   => "(lam ?x (+ (var ?x) 5))"
43 }
44
45 test_fn! { lambda_if_elim, rules(),
46   "(if (= (var a) (var b))
47       (+ (var a) (var a))
48       (+ (var a) (var b)))"
49   => "(+ (var a) (var b))"
50 }
```

```
51  // Returns a list of rewrite rules
52  fn rules() -> Vec<Rewrite<Lambda, LambdaAnalysis>> { vec![
53
54    // open term rules
55    rw!("if-true";  "(if  true ?then ?else)" => "?then"),
56    rw!("if-false"; "(if false ?then ?else)" => "?else"),
57    rw!("if-elim";  "(if (= (var ?x) ?e) ?then ?else)" => "?else
58        if ConditionEqual::parse("(let ?x ?e ?then)",
59                                  "(let ?x ?e ?else)")),
60    rw!("add-comm";  "(+ ?a ?b)"          => "(+ ?b ?a)"),
61    rw!("add-assoc"; "(+ (+ ?a ?b) ?c)" => "(+ ?a (+ ?b ?c))"),
62    rw!("eq-comm";   "(= ?a ?b)"          => "(= ?b ?a)"),
63
64    // substitution introduction
65    rw!("fix";      "(fix ?v ?e)" =>
66                    "(let ?v (fix ?v ?e) ?e)"),
67    rw!("beta";     "(app (lam ?v ?body) ?e)" =>
68                    "(let ?v ?e ?body)"),
69
70    // substitution propagation
71    rw!("let-app"; "(let ?v ?e (app ?a ?b))" =>
72                   "(app (let ?v ?e ?a) (let ?v ?e ?b))"),
73    rw!("let-add"; "(let ?v ?e (+   ?a ?b))" =>
74                   "(+   (let ?v ?e ?a) (let ?v ?e ?b))"),
75    rw!("let-eq";  "(let ?v ?e (=   ?a ?b))" =>
76                   "(=   (let ?v ?e ?a) (let ?v ?e ?b))"),
77    rw!("let-if";  "(let ?v ?e (if ?cond ?then ?else))" =>
78                   "(if (let ?v ?e ?cond)
79                        (let ?v ?e ?then)
80                        (let ?v ?e ?else))"),
81
82    // substitution elimination
83    rw!("let-const";     "(let ?v ?e ?c)" => "?c"
84        if is_const(var("?c"))),
85    rw!("let-var-same"; "(let ?v1 ?e (var ?v1))" => "?e"),
86    rw!("let-var-diff"; "(let ?v1 ?e (var ?v2))" => "(var ?v2)"
87        if is_not_same_var(var("?v1"), var("?v2"))),
88    rw!("let-lam-same"; "(let ?v1 ?e (lam ?v1 ?body))" =>
89                        "(lam ?v1 ?body)"),
90    rw!("let-lam-diff"; "(let ?v1 ?e (lam ?v2 ?body))" =>
91        ( CaptureAvoid {
92          fresh: var("?fresh"), v2: var("?v2"), e: var("?e"),
93          if_not_free: "(lam ?v2 (let ?v1 ?e ?body))"
94                        .parse().unwrap(),
95          if_free: "(lam ?fresh (let ?v1 ?e
96                        (let ?v2 (var ?fresh) ?body)))"
97                    .parse().unwrap(),
98        })
99        if is_not_same_var(var("?v1"), var("?v2"))),
100 ]}
```

Fig. 10. egg is generic over user-defined languages; here we define a language and rewrite rules for a lambda calculus partial evaluator. The provided define_language! macro (lines 1-22) allows the simple definition of a language as a Rust enum, automatically deriving parsing and pretty printing. A value of type Lambda is an e-node that holds either data that the user can inspect or some number of e-class children (e-class Ids).

Rewrite rules can also be defined succinctly (lines 51-100). Patterns are parsed as s-expressions: strings from the define_language! invocation (ex: fix, =, +) and data from the variants (ex: false, 1) parse as operators or terms; names prefixed by "?" parse as pattern variables.

Some of the rewrites made are conditional using the "left => right if cond" syntax. The if-elim rewrite on line 57 uses egg's provided ConditionEqual as a condition, only applying the right-hand side if the e-graph can prove the two argument patterns equivalent. The final rewrite, let-lam-diff, is dynamic to support capture avoidance; the right-hand side is a Rust value that implements the Applier trait instead of a pattern. Figure 11 contains the supporting code for these rewrites.

We also show some of the tests (lines 27-50) from egg's lambda test suite. The tests proceed by inserting the term on the left-hand side, running egg's equality saturation, and then checking to make sure the right-hand side pattern can be found in the same e-class as the initial term.

- Checking for saturation, timeouts, and e-graph size limits.
- Orchestrating the read-phase, write-phase, rebuild system (Figure 4) that makes egg fast.
- Recording performance data at each iteration.
- Potentially coordinating rule execution so that expansive rules like associativity do not dominate the e-graph.
- Finally, extracting the best expression(s) according to a user-defined cost function.

egg provides these functionalities through its Runner and Extractor interfaces. Runners automatically detect saturation, and can be configured to stop after a time, e-graph size, or iterations limit. The equality saturation loop provided by egg calls rebuild, so users need not even know about egg's deferred invariant maintenance. Runners record various metrics about each iteration automatically, and the user can hook into this to report relevant data. Extractors select the optimal term from an e-graph given a user-defined, local cost function.[10] The two can be combined as well; users commonly record the "best so far" expression by extracting in each iteration.

Figure 10 also shows egg's test_fn! macro for easily creating tests (lines 27-50). These tests create an e-graph with the given expression, run equality saturation using a Runner, and check to make sure the right-hand pattern can be found in the same e-class as the initial expression.

## 5.2 Extensibility

For simple domains, defining a language and purely syntactic rewrites will suffice. However, our partial evaluator requires interpreted reasoning, so we use some of egg's more advanced features like e-class analyses and dynamic rewrites. Importantly, egg supports these extensibility features as a library: the user need not modify the e-graph or egg's internals.

Figure 11 shows the remainder of the code for our lambda calculus partial evaluator. It uses an e-class analysis (LambdaAnalysis) to track free variables and constants associated with each e-class. The implementation of the e-class analysis is in Lines 11-50. The e-class analysis invariant guarantees that the analysis data contains an over-approximation of free variables from terms represented in that e-class. The analysis also does constant folding (see the make and modify methods). The let-lam-diff rewrite (Line 90, Figure 10) uses the CaptureAvoid (Lines 81-100, Figure 11) dynamic right-hand side to do capture-avoiding substitution only when necessary based on the free variable information. The conditional rewrites from Figure 10 depend on the conditions is_not_same_var and is_var (Lines 68-74, Figure 11) to ensure correct substitution.

egg is extensible in other ways as well. As mentioned above, Extractors are parameterized by a user-provided cost function. Runners are also extensible with user-provided rule schedulers that can control the behavior of potentially troublesome rewrites. In typical equality saturation, each rewrite is searched for and applied each iteration. This can cause certain rewrites, commonly associativity or distributivity, to dominate others and make the search space less productive. Applied in moderation, these rewrites can trigger other rewrites and find greatly improved expressions, but they can also slow the search by exploding the e-graph exponentially in size. By default, egg uses the built-in backoff scheduler that identifies rewrites that are matching in exponentially-growing locations and temporarily bans them. We have observed that this greatly reduced run time (producing the same results) in many settings. egg can also use a conventional every-rule-every-time scheduler, or the user can supply their own.

---

[10]As mentioned in Section 4.3, extraction can be implemented as part of an e-class analysis. The separate Extractor feature is still useful for ergonomic and performance reasons.

```
1   type EGraph = egg::EGraph<Lambda, LambdaAnalysis>;
2   struct LambdaAnalysis;
3   struct FC {
4     free: HashSet<Id>,      // our analysis data stores free vars
5     constant: Option<Lambda>, // and the constant value, if any
6   }
7
8   // helper function to make pattern meta-variables
9   fn var(s: &str) -> Var { s.parse().unwrap() }
10
11  impl Analysis<Lambda> for LambdaAnalysis {
12    type Data = FC; // attach an FC to each eclass
13    // merge implements semilattice join by joining into `to`
14    // returning true if the `to` data was modified
15    fn merge(&self, to: &mut FC, from: FC) -> bool {
16      let before_len = to.free.len();
17      // union the free variables
18      to.free.extend(from.free.iter().copied());
19      if to.constant.is_none() && from.constant.is_some() {
20        to.constant = from.constant;
21        true
22      } else {
23        before_len != to.free.len()
24      }
25    }
26
27    fn make(egraph: &EGraph, enode: &Lambda) -> FC {
28      let f = |i: &Id| egraph[*i].data.free.iter().copied();
29      let mut free = HashSet::default();
30      match enode {
31        Use(v) => { free.insert(*v); }
32        Let([v, a, b]) => {
33          free.extend(f(b)); free.remove(v); free.extend(f(a));
34        }
35        Lambda([v, b]) | Fix([v, b]) => {
36          free.extend(f(b)); free.remove(v);
37        }
38        _ => enode.for_each_child(
39              |c| free.extend(&egraph[c].data.free)),
40      }
41      FC { free: free, constant: eval(egraph, enode) }
42    }
43
44    fn modify(egraph: &mut EGraph, id: Id) {
45      if let Some(c) = egraph[id].data.constant.clone() {
46        let const_id = egraph.add(c);
47        egraph.union(id, const_id);
48      }
49    }
50  }
```

```
51  // evaluate an enode if the children have constants
52  // Rust's `?` extracts an Option, early returning if None
53  fn eval(eg: &EGraph, enode: &Lambda) -> Option<Lambda> {
54    let c = |i: &Id| eg[*i].data.constant.clone();
55    match enode {
56      Num(_) | Bool(_) => Some(enode.clone()),
57      Add([x, y]) => Some(Num(c(x)? + c(y)?)),
58      Eq([x, y]) => Some(Bool(c(x)? == c(y)?)),
59      _ => None,
60    }
61  }
62
63  // Functions of this type can be conditions for rewrites
64  trait ConditionFn = Fn(&mut EGraph, Id, &Subst) -> bool;
65
66  // The following two functions return closures of the
67  // correct signature to be used as conditions in Figure 10.
68  fn is_not_same_var(v1: Var, v2: Var) -> impl ConditionFn {
69    |eg, _, subst| eg.find(subst[v1]) != eg.find(subst[v2])
70  }
71  fn is_const(v: Var) -> impl ConditionFn {
72    // check the LambdaAnalysis data
73    |eg, _, subst| eg[subst[v]].data.constant.is_some()
74  }
75
76  struct CaptureAvoid {
77    fresh: Var, v2: Var, e: Var,
78    if_not_free: Pattern<Lambda>, if_free: Pattern<Lambda>,
79  }
80
81  impl Applier<Lambda, LambdaAnalysis> for CaptureAvoid {
82    // Given the egraph, the matching eclass id, and the
83    // substitution generated by the match, apply the rewrite
84    fn apply_one(&self, egraph: &mut EGraph,
85                 id: Id, subst: &Subst) -> Vec<Id>
86    {
87      let (v2, e) = (subst[self.v2], subst[self.e]);
88      let v2_free_in_e = egraph[e].data.free.contains(&v2);
89      if v2_free_in_e {
90        let mut subst = subst.clone();
91        // make a fresh symbol using the eclass id
92        let sym = Lambda::Symbol(format!("_{}", id).into());
93        subst.insert(self.fresh, egraph.add(sym));
94        // apply the given pattern with the modified subst
95        self.if_free.apply_one(egraph, id, &subst)
96      } else {
97        self.if_not_free.apply_one(egraph, id, &subst)
98      }
99    }
100 }
```

Fig. 11. Our partial evaluator example highlights three important features egg provides for extensibility: e-class analyses, conditional rewrites, and dynamic rewrites.

The LambdaAnalysis type, which implements the Analysis trait, represents the e-class analysis. Its associated data (FC) stores the constant term from that e-class (if any) and an over-approximation of the free variables used by terms in that e-class. The constant term is used to perform constant folding. The merge operation implements the semilattice join, combining the free variable sets and taking a constant if one exists. In make, the analysis computes the free variable sets based on the e-node and the free variables of its children; the eval generates the new constants if possible. The modify hook of Analysis adds the constant to the e-graph.

Some of the conditional rewrites in Figure 10 depend on conditions defined here. Any function with the correct signature may serve as a condition.

The CaptureAvoid type implements the Applier trait, allowing it to serve as the right-hand side of a rewrite. CaptureAvoid takes two patterns and some pattern variables. It checks the free variable set to determine if a capture-avoiding substitution is required, applying the if_free pattern if so and the if_not_free pattern otherwise.

## 5.3 Efficiency

egg's novel *rebuilding* algorithm (Section 3) combined with systems programming best practices makes e-graphs—and the equality saturation use case in particular—more efficient than prior tools.

egg is implemented in Rust, giving the compiler freedom to specialize and inline user-written code. This is especially important as egg's generic nature leads to tight interaction between library code (e.g., searching for rewrites) and user code (e.g., comparing operators). egg is designed from the ground up to use cache-friendly, flat buffers with minimal indirection for most internal data structures. This is in sharp contrast to traditional representations of e-graphs [Detlefs et al. 2005; Nelson 1980] that contains many tree- and linked list-like data structures. egg additionally compiles patterns to be executed by a small virtual machine [de Moura and Bjørner 2007], as opposed to recursively walking the tree-like representation of patterns.

Aside from deferred rebuilding, egg's equality saturation algorithm leads to implementation-level performance enhancements. Searching for rewrite matches, which is the bulk of running time, can be parallelized thanks to the phase separation. Either the rules or e-classes could be searched in parallel. Furthermore, the once-per-iteration frequency of rebuilding allows egg to establish other performance-enhancing invariants that hold during the read-only search phase. For example, egg sorts e-nodes within each e-class to enable binary search, and also maintains a cache mapping function symbols to e-classes that contain e-nodes with that function symbol.

Many of egg's extensibility features can also be used to improve performance. As mentioned above, rule scheduling can lead to great performance improvement in the face of "expansive" rules that would otherwise dominate the search space. The Runner interface also supports user hooks that can stop the equality saturation after some arbitrary condition. This can be useful when using equality saturation to prove terms equal; once they are unified, there is no point in continuing. egg's Runners also support batch simplification, where multiple terms can be added to the initial e-graph before running equality saturation. If the terms are substantially similar, both rewriting and any e-class analyses will benefit from the e-graph's inherent structural deduplication. The case study in Section 6.1 uses batch simplification to achieve a large speedup with simplifying similar expressions.

## 6 CASE STUDIES

This section relates three independently-developed, published projects from diverse domains that incorporated egg as an easy-to-use, high-performance e-graph implementation. In all three cases, the developers had first rolled their own e-graph implementations. egg allowed them to delete code, gain performance, and in some cases dramatically broaden the project's scope thanks to egg's speed and flexibility. In addition to gaining performance, all three projects use egg's novel extensibility features like e-class analyses and dynamic/conditional rewrites.

## 6.1 Herbie: Improving Floating Point Accuracy

Herbie automatically improves accuracy for floating-point expressions, using random sampling to measure error, a set of rewrite rules for generating program variants, and algorithms that prune and combine program variants to achieve minimal error. Herbie received PLDI 2015's Distinguished Paper award [Panchekha et al. 2015] and has been continuously developed since then, sporting hundreds of Github stars, hundreds of downloads, and thousands of users on its online version. Herbie uses e-graphs for algebraic simplification of mathematical expressions, which is especially important for avoiding floating-point errors introduced by cancellation, function inverses, and redundant computation.

Until our case study, Herbie used a custom e-graph implementation written in Racket (Herbie's implementation language) that closely followed traditional e-graph implementations. With timeouts disabled, e-graph-based simplification consumed the vast majority of Herbie's run time. As a fix, Herbie sharply limits the simplification process, placing a size limit on the e-graph itself and a time limit on the whole procedure. When the timeout is exceeded, simplification fails altogether. Furthermore, the Herbie authors knew of several features that they believed would improve Herbie's output but could not be implemented because they required more calls to simplification and would thus introduce unacceptable slowdowns. Taken together, slow simplification reduced Herbie's performance, completeness, and efficacy.

We implemented a egg simplification backend for Herbie. The egg backend is over 3000× faster than Herbie's initial simplifier and is now used by default as of Herbie 1.4. Herbie has also backported some of egg's features like batch simplification and rebuilding to its e-graph implementation (which is still usable, just not the default), demonstrating the portability of egg's conceptual improvements.

*6.1.1 Implementation.* Herbie is implemented in Racket while egg is in Rust; the egg simplification backend is thus implemented as a Rust library that provides a C-level API for Herbie to access via foreign-function interface (FFI). The Rust library defines the Herbie expression grammar (with named constants, numeric constants, variables, and operations) as well as the e-class analysis necessary to do constant folding. The library is implemented in under 500 lines of Rust.

Herbie's set of rewrite rules is not fixed; users can select which rewrites to use using command-line flags. Herbie serializes the rewrites to strings, and the egg backend parses and instantiates them on the Rust side.

Herbie separates exact and inexact program constants: exact operations on exact constants (such as the addition of two rational numbers) are evaluated and added to the e-graph, while operations on inexact constants or that yield inexact outputs are not. We thus split numeric constants in the Rust-side grammar between exact rational numbers and inexact constants, which are described by an opaque identifier, and transformed Racket-side expressions into this form before serializing them and passing them to the Rust driver. To evaluate operations on exact constants, we used the constant folding e-class analysis to track the "exact value" of each e-class.[11] Every time an operation e-node is added to the egg e-graph, we check whether all arguments to that operation have exact value (using the analysis data), and if so do rational number arithmetic to evaluate it. The e-class analysis is cleaner than the corresponding code in Herbie's implementation, which is a built-in pass over the entire e-graph.

*6.1.2 Results.* Our egg simplification backend is a drop-in replacement to the existing Herbie simplifier, making it easy to compare speed and results. We compare using Herbie's standard test suite of roughly 500 benchmarks, with timeouts disabled. Figure 12 shows the results. The egg simplification backend is over 3000× faster than Herbie's initial simplifier. This speedup eliminated Herbie's largest bottleneck: the initial implementation dominated Herbie's total run time at 98.1%, backporting egg improvements into Herbie cuts that to about half the total run time, and egg simplification takes under 5% of the total run time. Practically, the run time of Herbie's initial implementation was smaller, since timeouts cause tests failures when simplification takes too long. Therefore, the speedup also improved Herbie's completeness, as simplification now never times out.

Since incorporating egg into Herbie, the Herbie developers have backported some of egg's key performance improvements into the Racket e-graph implementation. First, batch simplification gives

---

[11]Herbie's rewrite rules guarantee that different exact values can never become equal; the semilattice join checks this invariant on the Rust side.
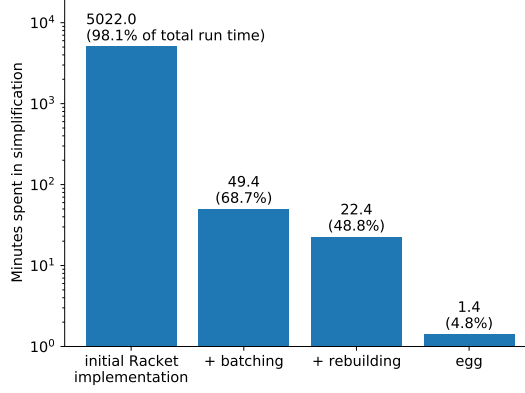
Fig. 12. Herbie sped up its expression simplification phase by adopting egg-inspired features like batched simplification and rebuilding into its Racket-based e-graph implementation. Herbie also supports using egg itself for additional speedup. Note that the y-axis is log-scale.

a large speedup because Herbie simplifies many similar expressions. When done simultaneously in one equality saturation, the e-graph's structural sharing can massively deduplicate work. Second, deferring rebuilding (as discussed in Section 3) gives a further 2.2× speedup. As demonstrated in Figure 7, rebuilding offers an asymptotic speedup, so Herbie's improved implementation (and the egg backend as well) will scale better as the search size grows.

## 6.2 Spores: Optimizing Linear Algebra

Spores [Wang et al. 2020] is an optimizer for machine learning programs. It translates linear algebra (LA) expressions to relational algebra (RA), performs rewrites, and finally translates the result back to linear algebra. Each rewrite is built up from simple identities in relational algebra like the associativity of join. These relational identities express more fine-grained equality than textbook linear algebra identities, allowing Spores to discover novel optimizations not found by traditional optimizers based on LA identities. Spores performs holistic optimization, taking into account the complex interactions among factors like sparsity, common subexpressions, and fusible operators and their impact on execution time.

*6.2.1 Implementation.* Spores is implemented entirely in Rust using egg. egg empowers Spores to orchestrate the complex interactions described above elegantly and effortlessly. Spores works in three steps: first, it translates the input LA expression to RA; second, it optimizes the RA expression by equality saturation; finally, it translates the optimized RA expression back to LA. Since the translation between LA and RA is straightforward, we focus the discussion on the equality saturation step in RA. Spores represents a relation as a function from tuples to real numbers: $A : (a_1, a_2, ..., a_n) \rightarrow \mathbb{R}$. This is similar to the index notation in linear algebra, where a matrix A can be viewed as a function $\lambda i, j.A_{ij}$. A tuple is identified with a named record, e.g. $(1, 2) = \{a_1 : 1, a_2 : 2\} = \{a_2 : 2, a_1 : 1\}$, so that order in a tuple doesn't matter. There are just three operations on relations: join, union and aggregate. Join ($\otimes$) takes two relations and returns their natural join, multiplying the associated real number for joined tuples:

$$A \otimes B = \lambda \bar{a} \cup \bar{b}.A(\bar{a}) \times B(\bar{b})$$

$$A \oplus (B \oplus C) = \oplus(A, B, C) \qquad (\oplus \text{ is assoc. \& comm.}) \tag{1}$$

$$A \otimes (B \otimes C) = \otimes(A, B, C) \qquad (\otimes \text{ is assoc. \& comm.}) \tag{2}$$

$$A \otimes (B \oplus C) = A \otimes B \oplus A \otimes C \qquad (\otimes \text{ distributes over } \oplus) \tag{3}$$

$$\sum_i (A \oplus B) = \sum_i A \oplus \sum_i B \tag{4}$$

$$\sum_i \sum_j A = \sum_{i,j} A \tag{5}$$

$$A \otimes \sum_i B = \sum_i (A \otimes B) \qquad (\text{requires } i \notin A) \tag{6}$$

$$\sum_i A = A \otimes \text{dimension}(i) \qquad (\text{requires } i \notin A) \tag{7}$$

Fig. 13.  RA equality rules $R_{EQ}$.

Here $\bar{a}$ is the set of field names for the records in $A$. In RA terminology, $\bar{a}$ is the *schema* of $A$. Union ($\oplus$) is a join in disguise: it also performs natural join on its two arguments, but adds the associated real instead of multiplying it:

$$A \oplus B = \lambda \bar{a} \cup \bar{b}.A(\bar{a}) + B(\bar{b})$$

Finally, aggregate ($\Sigma$) sums its argument along a given dimension. It coincides precisely with the "sigma notation" in mathematics:

$$\sum_{a_i} A = \lambda \bar{a} - a_i. \sum_{a_i} A(\bar{a})$$

The RA identities, presented in Figure 13, are also simple and intuitive. The notation $i \notin A$ means $i$ is not in the schema of $A$, and $dim(i)$ is the size of dimension $i$ (e.g. length of rows in a matrix). In Equation 6, when $i \in A$, we first rename every $i$ to a fresh variable $i'$ in $B$, which gives us: $A \otimes \sum_i B = \sum_{i'} (A \otimes B[i \to i'])$. In addition to these equalities, Spores also supports replacing expressions with fused operators. For example, $(X - UV)^2$ can be replaced by $sqloss(X, U, V)$ which streams values from $X, U, V$ and computes the result without creating intermediate matrices. Each of these fused operators is encoded with a simple identity in egg.

Note that Equation 6 requires a way to store the schema of every expression during optimization. Spores uses an e-class analysis to annotate e-classes with the appropriate schema. It also leverages the e-class analysis for cost estimation, using a conservative cost model that overapproximates. As a result, equivalent expressions may have different cost estimates. The merge operation on the analysis data takes the lower cost, incrementally improving the cost estimate. Finally, Spores' e-class analysis also performs constant folding. As a whole, the e-class analysis is a composition of three smaller analyses in a similar style to the composition of lattices in abstract interpretation.

*6.2.2 Results.* Spores is integrated into Apache SystemML [Boehm 2019] in a prototype, where it is able to derive all of 84 hand-written rules and heuristics for sum-product optimization. It also discovered novel rewrites that contribute to 1.2× to 5× speedup in end-to-end experiments. With greedy extraction, all compilations completed within a second.

## 6.3  Szalinski: Decompiling CAD into Structured Programs

Several tools have emerged that reverse engineer high level Computer Aided Design (CAD) models from polygon meshes and voxels [Du et al. 2018; Ellis et al. 2018; Nandi et al. 2018; Sharma et al.
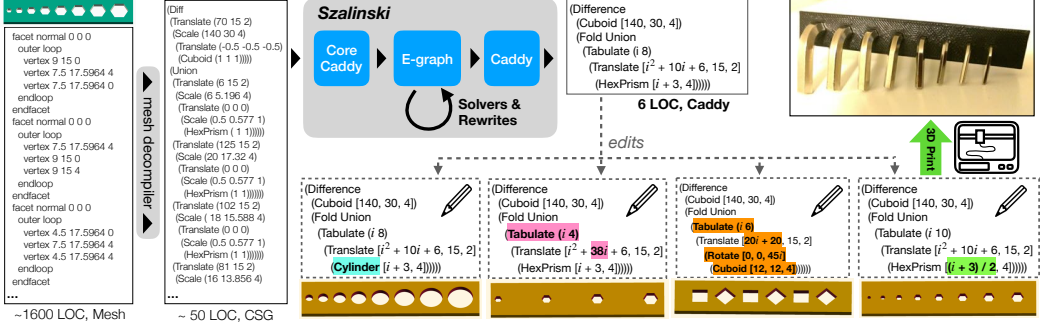
Fig. 14. (Figure from Nandi et. al. [Nandi et al. 2020]) Existing mesh decompilers turn triangle meshes into flat, computational solid geometry (CSG) expressions. Szalinski [Nandi et al. 2020] takes in these CSG expressions in a format called Core Caddy, and it synthesizes smaller, structured programs in language called Caddy that is enriched with functional-style features. This can ease customization by simplifying edits: small, mostly local changes yield usefully different models. The photo shows the 3D printed hex wrench holder after customizing hole sizes. Szalinski is powered by egg's extensible equality saturation, relying on its high performance, e-class analyses, and dynamic rewrites.

2017; Tian et al. 2019]. The output of these tools are constructive solid geometry (CSG) programs. A CSG program is comprised of 3D solids like cubes, spheres, cylinders, affine transformations like scale, translate, rotate (which take a 3D vector and a CSG expression as arguments), and binary operators like union, intersection, and difference that combine CSG expressions. For repetitive models like a gear, CSG programs can be too long and therefore difficult to comprehend. A recent tool, Szalinski [Nandi et al. 2020], extracts the inherent structure in the CSG outputs of mesh decompilation tools by automatically inferring maps and folds (Figure 14). Szalinski accomplished this using egg's extensible equality saturation system, allowing it to:

- Discover structure using loop rerolling rules. This allows Szalinski to infer functional patterns like Fold, Map2, Repeat and Tabulate from flat CSG inputs.
- Identify equivalence among CAD terms that are expressed as different expressions by mesh decompilers. Szalinski accomplishes this by using CAD identities. An example of one such CAD identity in Szalinski is $e \leftrightarrow rotate$ [0 0 0] $e$. This implies that any CAD expression $e$ is equivalent to a CAD expression that applies a rotation by zero degrees about x, y, and z axes to $e$.
- Use external solvers to speculatively add potentially profitable expressions to the e-graph. Mesh decompilers often generate CSG expressions that order and/or group list elements in non-intuitive ways. To recover structure from such expressions, a tool like Szalinski must be able to reorder and regroup lists that expose any latent structure.

*6.3.1 Implementation.* Even though CAD is different from traditional languages targeted by programming language techniques, egg supports Szalinski's CAD language in a straightforward manner. Szalinski uses purely syntactic rewrites to express CAD identities and some loop rerolling rules (like inferring a Fold from a list of CAD expressions). Critically, however, Szalinski relies on egg's dynamic rewrites and e-class analysis to infer functions for lists.

Consider the flat CSG program in Figure 15b. A structure finding rewrite first rewrites the flat list of Unions to:

```
(Fold Union (Map2 Translate [(0 0 0) (2 0 0) ...] (Repeat Cube 5)))
```
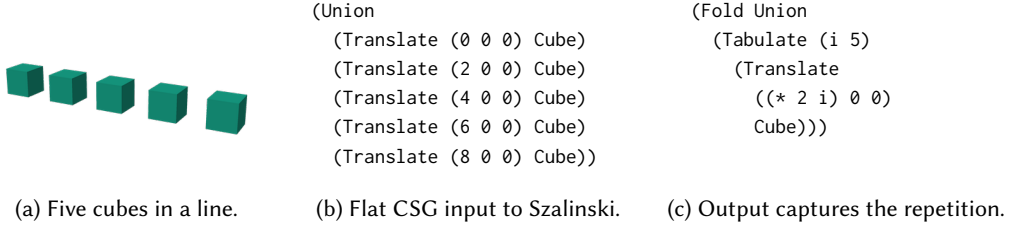
```
                                  (Union                          (Fold Union
                                    (Translate (0 0 0) Cube)        (Tabulate (i 5)
                                    (Translate (2 0 0) Cube)          (Translate
                                    (Translate (4 0 0) Cube)            ((* 2 i) 0 0)
                                    (Translate (6 0 0) Cube)            Cube)))
                                    (Translate (8 0 0) Cube))
```

(a) Five cubes in a line.        (b) Flat CSG input to Szalinski.        (c) Output captures the repetition.

Fig. 15. Szalinski integrates solvers into egg's equality saturation as a dynamic rewrite. The solver-backed rewrites can transform repetitive lists into `Tabulate` expressions that capture the repetitive structure.

The list of vectors is stored as `Cons` elements (sugared above for brevity). Szalinski uses an e-class analysis to track the accumulated lists in a similar style to constant folding. Then, a dynamic rewrite uses an arithmetic solver to rewrite the concrete list of 3D vectors in the analysis data to `(Tabulate (i 5) (* 2 i))`. A final set of syntactic rewrites can hoist the `Tabulate`, yielding the result on the right of Figure 15. Thanks to the set of syntactic CAD rewrites, this structure finding even works in the face of CAD identities. For example, the original program may omit the no-op `Translate (0 0 0)`, even though it is necessary to see repetitive structure.

In many cases, the repetitive structure of input CSG expression is further obfuscated because subexpressions may appear in arbitrary order. For these inputs, the arithmetic solvers must first reorder the expressions to find a closed form like a `Tabulate` as shown in Figure 15. However, reordering a list does not preserve equivalence, so adding it to the e-class of the concrete list would be unsound. Szalinski therefore introduces *inverse transformations*, a novel technique that allows solvers to speculatively reorder and regroup list elements to find a closed form. The solvers annotate the potentially profitable expression with the permutation or grouping that led to the successful discovery of the closed form. Later in the rewriting process, syntactic rewrites eliminate the inverse transformations when possible (e.g., reordering lists under a `Fold Union` can be eliminated). egg supported this novel technique without modification.

*6.3.2 Results.* Szalinski's initial protoype used a custom e-graph written in OCaml. Anecdotally, switching to egg removed most of the code, eliminated bugs, facilitated the key contributions of solver-backed rewrites and inverse transformations, and made the tool about 1000× faster. egg's performance allowed a shift from running on small, hand-picked examples to a comprehensive evaluation on over 2000 real-world models from a 3D model sharing forum [Nandi et al. 2020].

## 7  RELATED WORK

*Term Rewriting.* Term rewriting [Dershowitz and Jouannaud 1990] has been used widely to facilitate equational reasoning for program optimizations [Boyle et al. 1996; van den Brand et al. 2002; Visser et al. 1998]. A term rewriting system applies a database of semantics preserving rewrites or axioms to an input expression to get a new expression, which may, according to some cost function, be more profitable compared to the input. Rewrites are typically symbolic and have a left hand side and a right hand side. To apply a rewrite to an expression, a rewrite system implements pattern matching—if the left hand side of a rewrite rule matches with the input expression, the system computes a substitution which is then applied to the right-hand side of the rewrite rule. Upon applying a rewrite rule, a rewrite system typically replaces the old expression by the new expression. This can lead to the *phase ordering* problem— it makes it impossible to apply a rewrite to the old expression in the future which could have led to a more optimal result.

*E-graphs and E-matching.* E-graph were originally proposed several decades ago as an efficient data structure for maintaining congruence closure [Kozen 1977; Nelson 1980; Nelson and Oppen 1980]. E-graphs continue to be a critical component in successful SMT solvers where they are used for combining satisfiability theories by sharing equality information [De Moura and Bjørner 2008]. A key difference between past implementations of e-graphs and egg's e-graph is our novel rebuilding algorithm that maintains invariants only at certain critical points (Section 3). This makes egg more efficient for the purpose of equality saturation. egg implements the pattern compilation strategy introduced by de Moura et al. [de Moura and Bjørner 2007] that is used in state of the art theorem provers [De Moura and Bjørner 2008]. Some provers [De Moura and Bjørner 2008; Detlefs et al. 2005] propose optimizations like mod-time, pattern-element and inverted-path-index to find new terms and relevant patterns for matching, and avoid redundant matches. So far, we have found egg to be faster than several prior e-graph implementations even without these optimizations. They are, however, compatible with egg's design and could be explored in the future. Another key difference is egg's powerful e-class analysis abstraction and flexible interface. They empower the programmer to easily leverage e-graphs for problems involving complex semantic reasoning.

*Congruence Closure.* Our rebuilding algorithm is similar to the congruence closure algorithm presented by Downey et al. [1980]. The contribution of rebuilding is not *how* it restores the e-graph invariants but *when*; it gives the client the ability to specialize invariant restoration to a particular workload like equality saturation. Their algorithm also features a worklist of merges to be processed further, but it is offline, i.e., the algorithm processes a given set of equalities and outputs the set of equalities closed over congruence. Rebuilding is adapted to the online e-graph (and equality saturation) setting, where rewrites frequently examine the current set of equalities and assert new ones. Rebuilding additionally propagates e-class analysis facts (Section 4.1). Despite these differences, the core algorithms algorithms are similar enough that theoretical results on offline performance characteristics [Downey et al. 1980] apply to both. We do not provide theoretical analysis of rebuilding for the online setting; it is likely highly workload dependent.

*Superoptimization and Equality Saturation.* The Denali [Joshi et al. 2002] superoptimizer first demonstrated how to use e-graphs for optimized code generation as an alternative to hand-optimized machine code and prior exhaustive approaches [Massalin 1987], both of which were less scalable. The inputs to Denali are programs in a C-like language from which it produces assembly programs. Denali supported three types of rewrites—arithmetic, architectural, and program-specific. After applying these rewrites till saturation, it used architectural description of the hardware to generate constraints that were solved using a SAT solver to output a near-optimal program. While Denali's approach was a significant improvement over prior work, it was intended to be used on straight line code only and therefore, did not apply to large real programs.

Equality saturation [Stepp et al. 2011; Tate et al. 2009] developed a compiler optimization phase that works for complex language constructs like loops and conditionals. The first equality saturation paper used an intermediate representation called Program Expression Graphs (PEGs) to encode loops and conditionals. PEGs have specialized nodes that can represent infinite sequences, which allows them to represent loops. It uses a global profitability heuristic for extraction which is implemented using a pseudo-boolean solver. Recently, [Premtoon et al. 2020] uses PEGs for code search. egg can support PEGs as a user-defined language, and thus their technique could be ported.

## 8 CONCLUSION

We presented two new techniques, rebuilding and e-class analysis, that make equality saturation fast and extensible enough for a new family of applications. Rebuilding is a new way to amortize the cost of maintaining the e-graph's data structure invariants, specializing the e-graph to the equality

saturation workload. E-class analysis is a general framework allowing for interpreted reasoning beyond what purely syntactic rewrites can provide.

We implemented both of these techniques in egg, a reusable, extensible, and efficient e-graph library. egg is generic over the user-defined language, which allowed focused effort on optimization and efficiency while obviating the need for ad hoc e-graph implementations and manipulations. Our case studies show that equality saturation can now scale further and be used more flexibly than before; egg provided new functionality and large speedups. We believe that these contributions position equality saturation as a powerful toolkit for program synthesis and optimization.

## ACKNOWLEDGMENTS

## REFERENCES

Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. 1999. Graph Transformation for Specification and Programming. *Sci. Comput. Program.* 34, 1 (April 1999), 1–54. https://doi.org/10.1016/S0167-6423(98)00023-9

Matthias Boehm. 2019. Apache SystemML. *Encyclopedia of Big Data Technologies* (2019), 81–86. https://doi.org/10.1007/978-3-319-77525-8_187

James M. Boyle, Terence J. Harmer, and Victor L. Winter. 1996. The TAMPR Program Transformation System: Simplifying the Development of Numerical Software. In *Modern Software Tools for Scientific Computing, SciTools 1996, Oslo, Norway, September 16-18, 1996*, Erlend Arge, Are Magnus Bruaset, and Hans Petter Langtangen (Eds.). Birkhäuser, 353–372. https://doi.org/10.1007/978-1-4612-1986-6_17

Martin Davis and Hilary Putnam. 1960. A Computing Procedure for Quantification Theory. *J. ACM* 7, 3 (July 1960), 201–215. https://doi.org/10.1145/321033.321034

Leonardo de Moura and Nikolaj Bjørner. 2007. Efficient E-Matching for SMT Solvers. In *Automated Deduction – CADE-21*, Frank Pfenning (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 183–198.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

Nachum Dershowitz. 1993. *A taste of rewrite systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 199–228. https://doi.org/10.1007/3-540-56883-2_11

Nachum Dershowitz and Jean-Pierre Jouannaud. 1990. Rewrite Systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, Jan van Leeuwen (Ed.). Elsevier and MIT Press, 243–320. https://doi.org/10.1016/b978-0-444-88074-1.50011-1

David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: A Theorem Prover for Program Checking. *J. ACM* 52, 3 (May 2005), 365–473. https://doi.org/10.1145/1066100.1066102

Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. 1980. Variations on the Common Subexpression Problem. *J. ACM* 27, 4 (Oct. 1980), 758–771. https://doi.org/10.1145/322217.322228

Tao Du, Jeevana Priya Inala, Yewen Pu, Andrew Spielberg, Adriana Schulz, Daniela Rus, Armando Solar-Lezama, and Wojciech Matusik. 2018. InverseCSG: automatic conversion of 3D models to CSG trees. 1–16. https://doi.org/10.1145/3272127.3275006

Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2018. Learning to Infer Graphics Programs from Hand-Drawn Images. In *Neural Information Processing Systems (NIPS)*.

Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 47–62.

Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: A Goal-directed Superoptimizer. *SIGPLAN Not.* 37, 5 (May 2002), 304–314. https://doi.org/10.1145/543552.512566

Dexter Kozen. 1977. Complexity of Finitely Presented Algebras. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing* (Boulder, Colorado, USA) *(STOC '77)*. Association for Computing Machinery, New York, NY, USA, 164–177. https://doi.org/10.1145/800105.803406

Henry Massalin. 1987. Superoptimizer: A Look at the Smallest Program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems* (Palo Alto, California, USA) *(ASPLOS II)*. IEEE Computer Society Press, Washington, DC, USA, 122–126. https://doi.org/10.1145/36206.36194

Chandrakana Nandi, James R. Wilcox, Pavel Panchekha, Taylor Blau, Dan Grossman, and Zachary Tatlock. 2018. Functional Programming for Compiling and Decompiling Computer-aided Design. *Proc. ACM Program. Lang.* 2, ICFP, Article 99 (July 2018), 31 pages. https://doi.org/10.1145/3236794

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

Charles Gregory Nelson. 1980. *Techniques for Program Verification*. Ph.D. Dissertation. Stanford, CA, USA. AAI8011683.

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

Robert Nieuwenhuis and Albert Oliveras. 2005. Proof-Producing Congruence Closure. In *Proceedings of the 16th International Conference on Term Rewriting and Applications* (Nara, Japan) *(RTA'05)*. Springer-Verlag, Berlin, Heidelberg, 453–468. https://doi.org/10.1007/978-3-540-32033-3_33

Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. *SIGPLAN Not.* 50, 6 (June 2015), 1–11. https://doi.org/10.1145/2813885.2737959

Varot Premtoon, James Koppel, and Armando Solar-Lezama. 2020. Semantic Code Search via Equational Reasoning. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1066–1082. https://doi.org/10.1145/3385412.3386001

Rust. [n.d.]. *Rust programming language*. https://www.rust-lang.org/

Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhransu Maji. 2017. CSGNet: Neural Shape Parser for Constructive Solid Geometry. *CoRR* abs/1712.08290 (2017). arXiv:1712.08290 http://arxiv.org/abs/1712.08290

Michael Stepp, Ross Tate, and Sorin Lerner. 2011. Equality-Based Translation Validator for LLVM. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 737–742.

Robert Endre Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM* 22, 2 (April 1975), 215–225. https://doi.org/10.1145/321879.321884

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. ACM, New York, NY, USA, 264–276. https://doi.org/10.1145/1480881.1480915

Yonglong Tian, Andrew Luo, Xingyuan Sun, Kevin Ellis, William T. Freeman, Joshua B. Tenenbaum, and Jiajun Wu. 2019. Learning to Infer and Execute 3D Shape Programs. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rylNH20qFQ

Mark van den Brand, Jan Heering, Paul Klint, and Pieter A. Olivier. 2002. Compiling language definitions: the ASF+SDF compiler. *ACM Trans. Program. Lang. Syst.* 24, 4 (2002), 334–368. https://doi.org/10.1145/567097.567099

Eelco Visser, Zine-El-Abidine Benaissa, and Andrew P. Tolmach. 1998. Building Program Optimizers with Rewriting Strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*, Matthias Felleisen, Paul Hudak, and Christian Queinnec (Eds.). ACM, 13–26. https://doi.org/10.1145/289423.289425

Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. 2020. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *Proceedings of the VLDB Endowment* (2020).

Chenming Wu, Haisen Zhao, Chandrakana Nandi, Jeffrey I. Lipton, Zachary Tatlock, and Adriana Schulz. 2019. Carpentry Compiler. *ACM Transactions on Graphics* 38, 6 (2019), Article No. 195. presented at SIGGRAPH Asia 2019.