Practical, Flexible Equality Saturation

Max Willsey

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2021

Reading Commitee:

Luis Ceze, Chair

Adriana Schulz

Zachary Tatlock

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Practical, Flexible Equality Saturation

Max Willsey

Chair of the Supervisory Committee:
Professor Luis Ceze
Computer Science & Engineering

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.

Contents

Acknowledgements

Thanks to some people.

Chapter 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 [Nel80, NO05] extend union-find [Tar75] 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* [JNR02, TSTL09, STL11, NWA$^+$20, PKSL20, WHL$^+$20, PSSWT15]. 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

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

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 [PSSWT15, TSTL09, WZN^{+}19]. These challenges limit equality saturation's practicality.

Equality Saturation Workload. ATPs frequently query and modify e-graphs and additionally require *backtracking* to undo modifications (e.g., in DPLL(T) [DP60]). 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 (Chapter 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 (Chapter 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 (Chapter 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 (Chapter 6). Where previous implementations existed, egg is orders of magnitude faster and offers more features.

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

Chapter 2

Background

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

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 [Nel80, DNS05, DMB08], e-graphs have also been used to power a program optimization technique called *equality saturation* [JNR02, TSTL09, STL11, NWA$^+$20, PKSL20, WHL$^+$20, PSSWT15].

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 2.1, an e-graph is a tuple* (U, M, H) *where:*

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

Figure 2.1: 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 $=$.

- *A union-find data structure [Tar75] U stores an equivalence relation (denoted with $\equiv_{\text{id}}$) 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., $a \equiv_{\text{id}} b$ iff $M[a]$ is the same set as $M[b]$. An e-class id a is said to refer to the e-class $M[\texttt{find}(a)]$.*
- *The hashcons[1] 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.[2] We use e-class id a and the e-class $M[\texttt{find}(a)]$ 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 $\texttt{find}(U, a) = \texttt{find}(U, b)$ iff $a \equiv_{\text{id}} b$. We omit the first argument of `find` where clear from context.*

- *An e-class id a is canonical iff $\texttt{find}(a) = a$.*
- *An e-node n is canonical iff $n = \texttt{canonicalize}(n)$, where*

  $\texttt{canonicalize}(f(a_1, a_2, ...)) = f(\texttt{find}(a_1), \texttt{find}(a_2), ...)$.

[1]We use the term *hashcons* to evoke the memoization technique, since both avoid creating new duplicates of existing objects.

[2]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.

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 $f(a_1, a_2, ...)$ represents a term $f(t_1, t_2, ...)$ if they have the same function symbol f and e-class $M[a_i]$ represents term t_i.*

When each e-class is a singleton (containing only one e-node), an e-graph is essentially a term graph with sharing. Figure 2.2a shows an e-graph that represents the expression $(a \times 2)/2$.

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

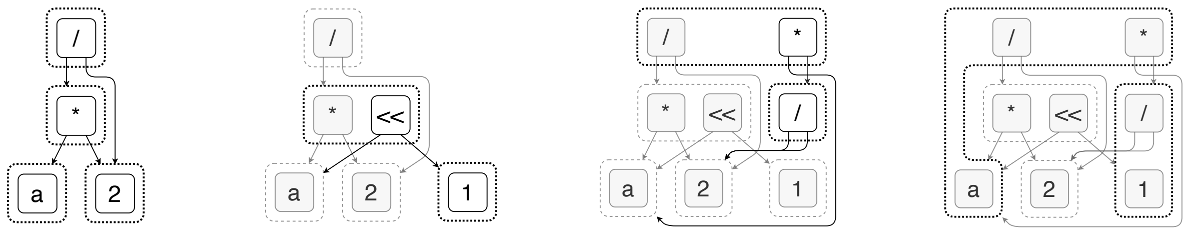
- *Over e-class ids: $a \equiv_{\mathsf{id}} b$ iff $\texttt{find}(a) = \texttt{find}(b)$.*
- *Over e-nodes: $n_1 \equiv_{\mathsf{node}} n_2$ iff e-nodes n_1, n_2 are in the same e-class, i.e., $\exists a.\ n_1, n_2 \in M[a]$.*
- *Over terms: $t_1 \equiv_{\mathsf{term}} t_2$ iff terms t_1, t_2 are represented 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 $a_i \equiv_{\mathsf{id}} b_i$. Let $\cong^*$ denote the congruence closure of $\equiv_{\mathsf{node}}$, i.e., the smallest superset of $\equiv_{\mathsf{node}}$ that is also a superset of $\cong$. Note that there may be two e-nodes such that $n_1 \cong^* n_2$ but $n_1 \ncong n_2$ and $n_1 \not\equiv_{\mathsf{node}} n_2$. 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 Chapter 3. These are collectively referred to as the *e-graph invariants.*



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

(b) Applied $x \times 2 \to x \ll 1$.

(c) Applied rewrite $(x \times y)/z \to x \times (y/z)$.

(d) Applied $x/x \to 1$ and $1 \times x \to x$.

Figure 2.2: 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 2.2d, 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.

Definition 2.6 (The Congruence Invariant) *The equivalence relation over e-nodes must be closed over congruence, i.e.,* $(\equiv_{\text{node}}) = (\cong^*)$. *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:*

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

If the hashcons invariant holds, then a procedure `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 $M[a] \cup M[b]$.

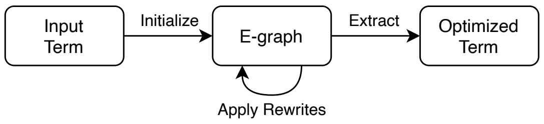
Both of these operations must take additional steps to maintain the congruence invariant. Invariant maintenance is discussed in Chapter 3.

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* [DNS05, dMB07] procedure for finding patterns in the e-graph. `ematch` takes a pattern term p with variable placeholders and returns a list of tuples (σ, c) where σ 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 $\ell \to r$ to an e-graph, `ematch` finds tuples (σ, 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 2.2 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 $x \times y \to y \times x$ 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.



```
def equality_saturation(expr, rewrites):
  egraph = initial_egraph(expr)

  while not egraph.is_saturated_or_timeout():

    for rw in rewrites:
      for (subst, eclass) in egraph.ematch(rw.lhs):
        eclass2 = egraph.add(rw.rhs.subst(subst))
        egraph.merge(eclass, eclass2)

  return egraph.extract_best()
```

Figure 2.3: Box diagram and pseudocode for equality saturation. Traditionally, equality saturation maintains the e-graph data structure invariants throughout the algorithm.

2.2 Equality Saturation

Term rewriting [Der93] is a time-tested approach for equational reasoning in program optimization [TSTL09, JNR02], theorem proving [DNS05, DMB08], and program transformation [AEH$^+$99]. 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: $(a \times 2)/2 \to (a \ll 1)/2$. 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* [TSTL09, STL11] is a technique to do this rewriting efficiently using an e-graph.

Figure 2.3 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 [WHL$^+$20, WZN$^+$19].

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 [JPT$^+$19], 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 [DMB08] 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 ($15\times$ faster), or only 0.52 seconds ($47\times$ faster) when using `egg`'s batched evaluation (Section 5.3).

Chapter 3

Rebuilding: A New Take on E-graph Invariant Maintenance

Traditionally [Nel80, DNS05], 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 [PSSWT15] 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

```
1  def add(enode):
2    enode = self.canonicalize(enode)
3    if enode in self.hashcons:
4      return self.hashcons[enode]
5    else:
6      eclass_id = self.new_singleton_eclass(enode)
7      for child in enode.children:
8        child.parents.add(enode, eclass_id)
9      self.hashcons[enode] = eclass_id
10     return eclass_id
11
12 def merge(id1, id2)
13   if self.find(id1) == self.find(id2):
14     return self.find(id1)
15   new_id = self.union_find.union(id1, id2)
16   # traditional egraph merge can be
17   # emulated by calling rebuild right after
18   # adding the eclass to the worklist
19   self.worklist.add(new_id)
20   return new_id
21
22 def canonicalize(enode)
23   new_ch = [self.find(e) for e in enode.children]
24   return mk_enode(enode.op, new_ch)
25
26 def find(eclass_id):
27   return self.union_find.find(eclass_id)
```

```
27 def rebuild():
28   while self.worklist.len() > 0:
29     # empty the worklist into a local variable
30     todo = take(self.worklist)
31     # canonicalize and deduplicate the eclass refs
32     # to save calls to repair
33     todo = { self.find(eclass) for eclass in todo }
34     for eclass in todo:
35       self.repair(eclass)
36
37 def repair(eclass):
38   # update the hashcons so it always points
39   # canonical enodes to canonical eclasses
40   for (p_node, p_eclass) in eclass.parents:
41     self.hashcons.remove(p_node)
42     p_node = self.canonicalize(p_node)
43     self.hashcons[p_node] = self.find(p_eclass)
44
45   # deduplicate the parents, noting that equal
46   # parents get merged and put on the worklist
47   new_parents = {}
48   for (p_node, p_eclass) in eclass.parents:
49     p_node = self.canonicalize(p_node)
50     if p_node in new_parents:
51       self.merge(p_eclass, new_parents[p_node])
52     new_parents[p_node] = self.find(p_eclass)
53   eclass.parents = new_parents
```

Figure 3.1: Pseudocode for the `add`, `merge`, `rebuild`, and supporting methods. In each method, `self` refers to the e-graph being modified.

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 **??** for comparison to [DST80]); but it uniquely allows the client to choose when to restore invariants in the context of a larger algorithm like equality saturation.

Figure 3.1 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 3.1 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.

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:

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

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 xs together: for $i \in [2, w]$, $\texttt{merge}(x_1, x_i)$.

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

With deferred rebuilding, however, the $w-1$ merges can all take place before congruence must be restored. Suppose the xs are all merged into an e-class c_x When `rebuild` finally is called, the only element in the deduplicated worklist is c_x. Calling `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 `repair`, eliminating the factor corresponding to the width of this group of terms. Figure 3.5 shows that the number calls to `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 3.1 *Since rebuilding only merges congruent nodes, the congruence closure $\cong^*$ is fixed even though $\equiv_{\mathsf{node}}$ changes. When $(\equiv_{\mathsf{node}}) = (\cong^*)$, congruence is restored. Note that both $\equiv_{\mathsf{node}}$ and*

$\cong^*$ *are finite. We therefore show that rebuilding causes* $\equiv_{\mathsf{node}}$ *to approach* $\cong^*$*. We define the set of incongruent e-node pairs as* $I = (\cong^*) \setminus (\equiv_{\mathsf{node}})$*; in other words,* $(n_1, n_2) \in I$ *if* $n_1 \cong^* n_2$ *but* $n_1 \not\equiv_{\mathsf{node}} n_2$.

Due to the additive nature of equality saturation, $\equiv_{\mathsf{node}}$ *only increases and therefore* I *is non-increasing. However, a call to* `repair` *inside the loop of* `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* `repair`*; in Figure 3.1,* W *corresponds to* `self.worklist` *plus the unprocessed portion of the* `todo` *local variable. We show that each call to* `repair` *decreases the tuple* $(|I|, |W|)$ *lexicographically until* $(|I|, |W|) = (0, 0)$*, and thus rebuilding terminates with* $(\equiv_{\mathsf{node}}) = (\cong^*)$.

Given an e-class c *from* W*,* `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* `repair` *did not add anything back.*

Since $(|I|, |W|)$ *decreases lexicographically,* $|W|$ *eventually reaches* 0*, so* `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,* `rebuild` *has effectively performed upward merging. By [Nel80, 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

```
def equality_saturation(expr, rewrites):
  egraph = initial_egraph(expr)

  while not egraph.is_saturated_or_timeout():


    # reading and writing is mixed
    for rw in rewrites:
      for (subst, eclass) in egraph.ematch(rw.lhs):

        # in traditional equality saturation,
        # matches can be applied right away
        # because invariants are always maintained
        eclass2 = egraph.add(rw.rhs.subst(subst))
        egraph.merge(eclass, eclass2)

        # restore the invariants after each merge
        egraph.rebuild()

  return egraph.extract_best()
```

(a) Traditional equality saturation alternates between searching and applying rules, and the e-graph maintains its invariants throughout.

```
def equality_saturation(expr, rewrites):
  egraph = initial_egraph(expr)

  while not egraph.is_saturated_or_timeout():
    matches = []

    # read-only phase, invariants are preserved
    for rw in rewrites:
      for (subst, eclass) in egraph.ematch(rw.lhs):
        matches.append((rw, subst, eclass))

    # write-only phase, temporarily break invariants
    for (rw, subst, eclass) in matches:
      eclass2 = egraph.add(rw.rhs.subst(subst))
      egraph.merge(eclass, eclass2)

    # restore the invariants once per iteration
    egraph.rebuild()

  return egraph.extract_best()
```

(b) `egg` splits equality saturation iterations into read and write phases. The e-graph invariants are not constantly maintained, but restored only at the end of each iteration by the `rebuild` method (Chapter 3).

Figure 3.2: Pseudocode for traditional and `egg`'s version of the equality saturation algorithm.

time that maximizes performance without limiting the application.

`egg` provides a modified equality saturation algorithm to take advantage of rebuilding. Figure 3.2 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.[1]

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

[1]Although the original equality saturation paper [TSTL09] does not have separate reading and writing phases, some e-graph implementations (like the one inside Z3 [DMB08]) do separate these phases as an implementation detail. Ours is the first algorithm to take advantage of this by deferring invariant maintenance.

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,[2] or saturation detection.

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 3.3 shows our how rebuilding speeds up congruence maintenance. Overall, our experiments show an aggregate $87.85\times$ speedup on congruence closure and $20.96\times$ speedup over the entire equality saturation algorithm. Figure 3.4 shows this speedup is asymptotic; the multiplicative

[2] An optimization introduced in Figure 5.2 that relies on an accurate count of how many times a rewrite was matched.

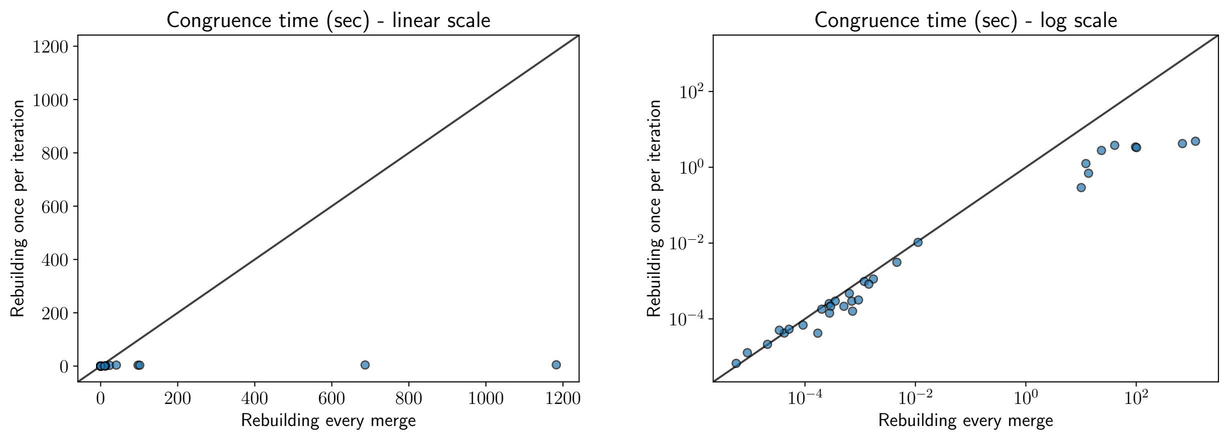
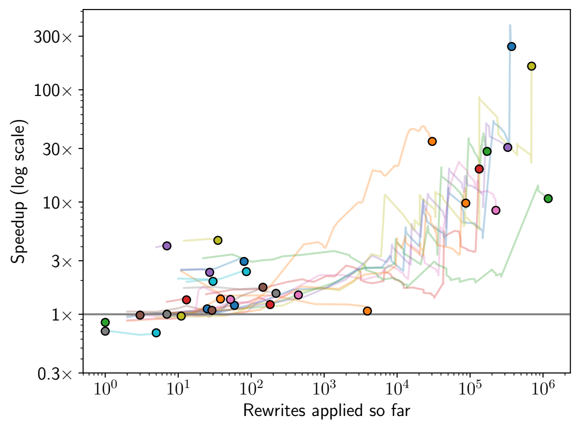


Figure 3.3: Rebuilding once per iteration—as opposed to after every merge—significantly speeds up congruence maintenance. Both plots show the same data: one point for each of the 32 tests. The diagonal line is $y = x$; points below the line mean deferring rebuilding is faster. In aggregate over all tests (using geometric mean), congruence is $87.85\times$ faster, and equality saturation is $20.96\times$ 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 3.4 demonstrates this in greater detail.



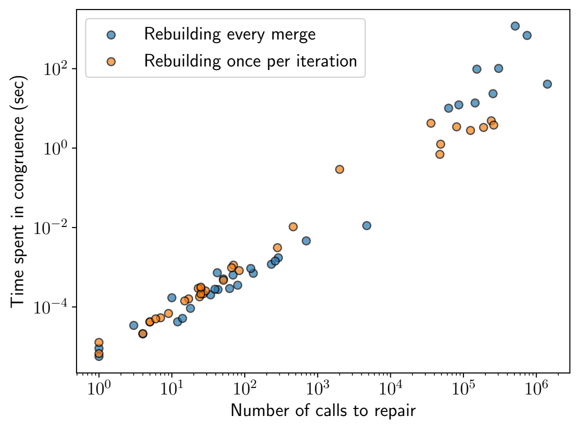


Figure 3.4: As more rewrites are applied, deferring rebuilding gives greater speedup. Each line represents a single test: each equality saturation iteration plots the cumulative rewrites applied so far against the multiplicative speedup of deferring rebuilding; the dot represents the end of that test. Both the test suite as a whole (the dots) and individual tests (the lines) demonstrate an asymptotic speedup that increases with the problem size.

Figure 3.5: 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.

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 Chapter 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 3.3, 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 3.5 shows that time spent in congruence is correlated with the number of calls to the `repair` methods.

The case study in Section 6.3 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.

Chapter 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 [TSTL09, STL11, PSSWT15] 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: Chapter 5 discusses the `egg` implementation and a complete example of a partial evaluator for the lambda calculus; Chapter 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:

$\mathsf{make}(n) \to d_c$	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.
$\mathsf{join}(d_{c_1}, d_{c_2}) \to d_c$	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.
$\mathsf{modify}(c) \to c'$	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., $\mathsf{modify}(\mathsf{modify}(c)) = \mathsf{modify}(c)$

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):

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

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 $\forall c, \forall n \in c, d_c \geq \mathsf{make}(n)$. 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 Chapter 3 to restore the analysis invariant as well as the congruence invariant. Figure 4.1 shows the necessary modifications to the rebuilding code from Figure 3.1.

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 $\mathsf{make}(n)$ 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 Chapter 3. Lines 37–44 of the `repair`

```
 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)
```

Figure 4.1: The pseudocode for maintaining the e-class analysis invariant is largely similar to how rebuilding maintains congruence closure (Chapter 3). Only lines 10–11, 17–18, and 37–44 are added. Grayed out or missing code is unchanged from Figure 3.1.

method (which `rebuild` on each element of the worklist) re-make and merge the analysis data 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: Note

```
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) }
}
```

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 γ 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 $x/x \to 1$ 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 Chapter 5 and case studies in Chapter 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 [PSSWT15, NWA+20], 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 [PSSWT15].

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 $\mathsf{make}(n)$ 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 $\mathsf{extract}(c) = f(\mathsf{extract}(c_1), \mathsf{extract}(c_2), ...)$. 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. **??** 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.

Chapter 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[1], for which we provide the complete source code (which few changes for readability) in Figure 5.1 and Figure 5.2. 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. Chapter 6 goes further, providing real-world case studies of published projects that have depended on `egg`.

[1]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.

`egg` is implemented in ~5000 lines of Rust,[2] including code, tests, and documentation. `egg` is open-source, well-documented, and distributed via Rust's package management system.[3] 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 5.1, 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 5.1, 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 5.2.

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:

- Checking for saturation, timeouts, and e-graph size limits.
- Orchestrating the read-phase, write-phase, rebuild system (Figure 3.1) that makes `egg` fast.
- Recording performance data at each iteration.

[2]Rust [Rus] 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.

[3]Source: `https://github.com/mwillsey/egg`. Documentation: `https://docs.rs/egg`. Package: `https://crates.io/crates/egg`.
This paper uses version 0.6 of `egg`.

```rust
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  }
```

```rust
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 ]}
```

Figure 5.1: `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 `Id`s).
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 5.2 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 pattern can be found in the same e-class as the initial term.

- 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. `Runner`s 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. `Runner`s record various metrics about each iteration automatically, and the user can hook into this to report relevant data. `Extractor`s select the optimal term from an e-graph given a user-defined, local cost function.[4] The two can be combined as well; users commonly record the "best so far" expression by extracting in each iteration.

Figure 5.1 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 5.2 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

[4]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.

```rust
type EGraph = egg::EGraph<Lambda, LambdaAnalysis>;
struct LambdaAnalysis;
struct FC {
  free: HashSet<Id>,    // our analysis data stores free vars
  constant: Option<Lambda>, // and the constant value, if any
}

// helper function to make pattern meta-variables
fn var(s: &str) -> Var { s.parse().unwrap() }

impl Analysis<Lambda> for LambdaAnalysis {
  type Data = FC; // attach an FC to each eclass
  // merge implements semilattice join by joining into ‘to‘
  // returning true if the ‘to‘ data was modified
  fn merge(&self, to: &mut FC, from: FC) -> bool {
    let before_len = to.free.len();
    // union the free variables
    to.free.extend(from.free.iter().copied());
    if to.constant.is_none() && from.constant.is_some() {
      to.constant = from.constant;
      true
    } else {
      before_len != to.free.len()
    }
  }

  fn make(egraph: &EGraph, enode: &Lambda) -> FC {
    let f = |i: &Id| egraph[*i].data.free.iter().copied();
    let mut free = HashSet::default();
    match enode {
      Use(v) => { free.insert(*v); }
      Let([v, a, b]) => {
        free.extend(f(b)); free.remove(v); free.extend(f(a));
      }
      Lambda([v, b]) | Fix([v, b]) => {
        free.extend(f(b)); free.remove(v);
      }
      _ => enode.for_each_child(
             |c| free.extend(&egraph[c].data.free)),
    }
    FC { free: free, constant: eval(egraph, enode) }
  }

  fn modify(egraph: &mut EGraph, id: Id) {
    if let Some(c) = egraph[id].data.constant.clone() {
      let const_id = egraph.add(c);
      egraph.union(id, const_id);
    }
  }
}
// evaluate an enode if the children have constants
// Rust’s ‘?‘ extracts an Option, early returning if None
fn eval(eg: &EGraph, enode: &Lambda) -> Option<Lambda> {
  let c = |i: &Id| eg[*i].data.constant.clone();
  match enode {
    Num(_) | Bool(_) => Some(enode.clone()),
    Add([x, y]) => Some(Num(c(x)? + c(y)?)),
    Eq([x, y]) => Some(Bool(c(x)? == c(y)?)),
    _ => None,
  }
}

// Functions of this type can be conditions for rewrites
trait ConditionFn = Fn(&mut EGraph, Id, &Subst) -> bool;

// The following two functions return closures of the
// correct signature to be used as conditions in Figure 5.1.
fn is_not_same_var(v1: Var, v2: Var) -> impl ConditionFn {
    |eg, _, subst| eg.find(subst[v1]) != eg.find(subst[v2])
}
fn is_const(v: Var) -> impl ConditionFn {
     // check the LambdaAnalysis data
    |eg, _, subst| eg[subst[v]].data.constant.is_some()
}

struct CaptureAvoid {
  fresh: Var, v2: Var, e: Var,
  if_not_free: Pattern<Lambda>, if_free: Pattern<Lambda>,
}

impl Applier<Lambda, LambdaAnalysis> for CaptureAvoid {
  // Given the egraph, the matching eclass id, and the
  // substitution generated by the match, apply the rewrite
  fn apply_one(&self, egraph: &mut EGraph,
               id: Id, subst: &Subst) -> Vec<Id>
  {
    let (v2, e) = (subst[self.v2], subst[self.e]);
    let v2_free_in_e = egraph[e].data.free.contains(&v2);
    if v2_free_in_e {
      let mut subst = subst.clone();
      // make a fresh symbol using the eclass id
      let sym = Lambda::Symbol(format!("_{}", id).into());
      subst.insert(self.fresh, egraph.add(sym));
      // apply the given pattern with the modified subst
      self.if_free.apply_one(egraph, id, &subst)
    } else {
      self.if_not_free.apply_one(egraph, id, &subst)
    }
  }
}
```

Figure 5.2: 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 5.1 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.

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 5.1) uses the `CaptureAvoid` (Lines 81-100, Figure 5.2) dynamic right-hand side to do capture-avoiding substitution only when necessary based on the free variable information. The conditional rewrites from Figure 5.1 depend on the conditions `is_not_same_var` and `is_var` (Lines 68-74, Figure 5.2) to ensure correct substitution.

`egg` is extensible in other ways as well. As mentioned above, `Extractor`s are parameterized by a user-provided cost function. `Runner`s 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.

5.3 Efficiency

`egg`'s novel *rebuilding* algorithm (Chapter 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 [Nel80, DNS05] that contains many tree- and linked list-like data structures. `egg` additionally compiles patterns to be executed by a small virtual machine [dMB07], 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 `Runner`s 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 6.3 uses batch simplification to achieve a large speedup with simplifying similar expressions.

Chapter 6

Case Studies

Case studies are good.

6.1 CAD Simplification

6.2 Rewrite Synthesis

6.3 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 [PSSWT15] 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 $3000\times$ 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.

6.3.1 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

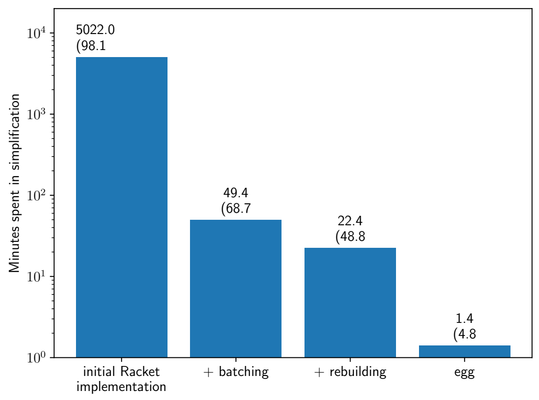


Figure 6.1: Herbie sped up its expression simplification phase by adopting `egg`-inspired features like batched simplification and rebuilding into its Racket-based e-graph implementation. Herbie also supports using `egg` itself for additional speedup. Note that the y-axis is log-scale.

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.[1] 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.

6.3.2 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 6.1 shows the results. The `egg` simplification backend is over $3000\times$ faster than Herbie's initial simplifier. This speedup eliminated Herbie's largest

[1]Herbie's rewrite rules guarantee that different exact values can never become equal; the semilattice join checks this invariant on the Rust side.

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 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 Chapter 3) gives a further $2.2\times$ speedup. As demonstrated in Figure 3.4, rebuilding offers an asymptotic speedup, so Herbie's improved implementation (and the `egg` backend as well) will scale better as the search size grows.

6.4 Tensat: Optimizing Deep Learning Computation Graphs

Deep learning frameworks and compilers have enabled diverse kinds of machine learning models to run efficiently on numerous compute platforms. Neural network models in these frameworks are typically represented as tensor computation graphs. To improve the runtime performance of a tensor graph, these frameworks perform various optimizations.

One of the most important optimizations is graph rewriting, which takes in a tensor graph g and a set of semantics-preserving graph rewrites R, and by applying rewrites to g seeks to find an semantically equivalent g' with lower cost according to some cost model. The current industry-standard approach adopted by most frameworks is to use a manually curated set of rewrite rules and rely on a heuristic strategy to determine the order in which to apply the rewrite rules. However, this approach often leads to sub-optimal results both due to the non-comprehensive set of rewrite rules, as well as the sub-optimal graph substitution heuristic [JPT+19, JTW+19].

	Search time (s)		**Runtime speedup (%)**	
	TASO	Tensat	TASO	Tensat
BERT	13.6	**1.4**	8.5	**9.2**
ResNeXt-50	25.3	**0.7**	5.5	**8.8**
NasNet-A	1226	**10.6**	1.9	**7.3**
NasRNN	177.3	**0.5**	45.4	**68.9**
Inception-v3	68.6	**5.1**	6.3	**10.0**
SqueezeNet	16.4	**0.3**	6.7	**24.5**
VGG-19	8.9	**0.4**	**8.9**	**8.9**

Table 6.1: Comparison of optimization time and runtime speedup of the optimized computation graphs over the original graphs, TASO [JPT+19] v.s. Tensat.

This paper aims to address the sub-optimality problem of graph rewrite strategies, while leveraging the existing rewrite rules generation technique [JPT+19]. Prior research has shown that searching for sequences of substitutions [JPT+19, JTW+19, FSWC20] outperforms heuristic approaches. However, both heuristic and search-based solutions rely on sequential application of substitutions. Since rewrites often depend on or enable one another, optimization depends heavily on the order in which rewrites are applied; this classically tricky problem is known in the compilers community as the "phase-ordering" or "rewrite-ordering" problem.

This paper presents Tensat, a tensor graph superoptimization framework that employs *equality saturation* [TSTL09, STL11, WWF+20], a recent technique that mitigates the phase-ordering problem by applying all possible rewrites at once. Equality saturation splits program optimization into two phases: *exploration* and *extraction*. The exploration phase uses a data structure called *e-graph* to compactly generate and store all rewritings of the input program. The exploration can continue until *saturation*, where the e-graph stores all possible ways to write the input program using a given set of rewrites. Finally, the extraction phase selects from the e-graph the equivalent program with the lowest cost according to a given cost model. The compact representation of the exponentially large search space using e-graphs enables extraction algorithms that can find the globally optimal equivalent program quickly.

Applying equality saturation to tensor graph rewriting requires non-trivial extensions in both the exploration and extraction phases. We extend the exploration phase to support complex, non-local rewrite rules that are necessary to produce highly efficient tensor graphs. Additionally, we introduce a novel method to filter out invalid subgraphs from an e-graph, which enables our extraction procedure based on Integer Linear Programming (ILP) to quickly find the optimal solution.

We evaluated Tensat on a number of well-known machine learning models executing on a GPU. As highlighted in Table 6.1, Tensat can synthesize optimized graphs that are up to 23% faster in runtime than state-of-the-art [JPT$^+$19], while reducing the optimization time by up to 300x. By having the e-graph compactly representing an exponential number of equivalent graphs, Tensat is able to cover a larger search space more efficiently than the sequential search methods. As a result, our search approach is both extremely effective and fast enough to be used as part of a normal complation flow.

6.4.1 Representations

This section describes how Tensat represents tensor computation graphs and rewrite rules.

Representing Tensor Computation Graphs

We use a representation based on the one in TASO [JPT$^+$19], with modifications to make it suitable for equality saturation. Table 6.2 shows the set of operators we consider. Each operator o_i corresponds to a node n_i in the graph; the node represents the output tensor of the operator. The nodes corresponding to the inputs of o_i are the children nodes of n_i. Each tensor computation graph is a DAG under this representation.

The formulations in equality saturation becomes simpler if a graph is single-rooted. Therefore, we combine all the final output nodes of a graph with *noop*s to make the graph single-rooted. The noop nodes do not have any actual operators associated with them, and they will not be altered during the exploration phase, so there is no side effects.

Representing Rewrite Rules

A rewrite rule for tensor computation graph specifies that some local subgraph pattern (*source pattern*) is equivalent to another subgraph pattern (*target pattern*). The input tensors to the source and target patterns are *variable nodes*, which can be substituted with any concrete nodes (or e-class in equality saturation) in the current graph. Each output tensor in the source pattern corresponds to an output tensor in the target pattern. The two corresponding output nodes are called a pair of *matched outputs*. A rewrite rule states the equivalence between each pair of matched outputs.

We represent each source (and target) pattern using symbolic expressions (S-exprs) with variables. Patterns with a single output is represented with an S-expr rooted on the output. Rewrite rules with such patterns are called *single-pattern rewrite rules*. Patterns with multiple outputs are represented as a list of S-exprs rooted on each output. Rewrite rules with multiple matched outputs are called *multi-pattern rewrite rules*. Figure 6.2 shows an example rewrite rule and its representation.

6.4.2 Exploration Phase

We initialize the e-graph with the original tensor computation graph. In each iteration of the exploration phase, we search for matches of all rewrite rules in the current e-graph, and add the target patterns and equivalence relations to the e-graph. This process continues until either the e-graph saturates or a user-specified limit (in terms of time, e-graph size, or number of iterations) is reached. Before applying a rewrite at a found match, we perform a *shape checking* to verify if the tensor shapes in the target pattern are compatible. This is necessary since some rewrite rules requires input tensor shapes to satisfy specific preconditions, in addition to the syntactic match. We perform shape checking in the same way as TASO [JPT+19].

Multi-Pattern Rewrite Rules Multi-pattern rewrite rules are an important type of rules for tensor graph superoptimization [JPT+19]. However, most equality saturation toolkits only support efficient search methods to find matches for single-pattern rewrite rules [WWF+20, dMB07]. We introduce an algorithm for applying multi-pattern rewrites, as shown in Algorithm 1. Our algorithm leverages the existing efficient search routine for single-pattern rewrites as a subroutine.

At the beginning of the exploration phase, we collect the set of unique S-exprs present in the source patterns of the rewrite rules after canonicalization. Here, if one S-expr can be transformed into another S-expr by variable renaming only, they will be mapped to the same canonicalized S-expr. In each iteration of the exploration phase, we use the single-pattern search subroutine to search for matches of the canonical S-exprs. Then for each multi-pattern rule, we take the Cartesian product of the matches found, decanonicalize the variable-to-e-class map into the original variables (using the variable renaming map stored during canonicalization), and check if the matches are compatible at the shared variables between the S-exprs (i.e., if the shared variables refer to the same e-class after the mapping). We apply the matches that are compatible.

In our experience, one feature of multi-pattern rules for tensor graph is that they can grow the e-graph extremely rapidly. Let's consider again the example rewrite rule in Figure 6.2. This rule can be matched with any two `matmul` nodes with a shared input (input_1). By applying this rule once on some match, a new `matmul` node will be created and added to the e-graph (the one on the RHS of Figure 6.2), which also has input_1 as its input. If the e-graph contains N `matmul` nodes that has some input_1 at the beginning, then after iteration 1, $\mathcal{O}(N^2)$ new `matmul` nodes sharing input_1 will be created. In iteration 2, each pair in these $\mathcal{O}(N^2)$ nodes will be a match, which will create $\mathcal{O}(N^4)$ new nodes. Such double exponential growth can quickly explode the e-graph.

Based on this feature, we set a separate limit k_{multi} on the number of iterations to apply the multi-pattern rules. After k_{multi} iterations, we only apply the single-pattern rules until saturation or some user-specified limit.

Algorithm 1 Applying multi-pattern rewrite rules

```
Input: starting e-graph $\mathcal{G}$, set of multi-pattern rewrite rules $\mathcal{R}_m$.
Output: updated e-graph $\mathcal{G}$.
 1: canonicalized S-expr $e_c$ = Set({})
 2: for rule $r \in \mathcal{R}_m$ do
 3:     for $i = 0, \dots, |r| - 1$ do                          ▷ $|r|$: #S-exprs in source pattern
 4:         ($e$, rename_map) = CANONICAL($r$.source[$i$])
 5:         $e_c$.insert($e$)
 6:         $r$.map[i] = rename_map
 7:     end for
 8: end for
 9: for iter = 0, …, MAX_ITER do
10:     $M$ = SEARCH($\mathcal{G}, e_c$)                          ▷ all matches for all patterns
11:     for rule $r \in \mathcal{R}_m$ do
12:         for $i = 0, \dots, |r| - 1$ do
13:             canonical matches mc$_i$ = $M$[$r$.source[i]]
14:             matches m$_i$ = DECANONICAL(mc$_i$, $r$.map[$i$])
15:         end for
16:         for $(\sigma_0, \dots, \sigma_{|r|-1}) \in \mathrm{m}_0 \times \cdots \times \mathrm{m}_{|r|-1}$ do
17:             if COMPATIBLE($(\sigma_0, \dots, \sigma_{|r|-1})$) then
18:                 APPLY($\mathcal{G}, r, \sigma_0, \dots, \sigma_{|r|-1}$)
19:             end if
20:         end for
21:     end for
22: end for
23: return $\mathcal{G}$
```

6.4.3 Extraction Phase

During extraction, the goal is to pick one e-node from each e-class in the e-graph to obtain an optimized graph. The optimized graph should minimize the total cost with respect to a given cost model. In tensor graph superoptimization, the cost model reflects the inference time taken by the graph.

Cost model We use the same cost model as TASO [JPT+19]. Each operator has a separate and independent cost, which is the measured runtime of that operator (with the specific input sizes and parameters) on hardware. The total cost of a graph is the sum of costs of each of its nodes. This cost model is suitable for GPUs, since GPUs typically run one operator at a time when executing a graph. Note that an operator can be a fused operator, consisting of multiple primitive operators, such as a fused convolution and ReLU.

Extraction Algorithms

Greedy extraction We first experiment with a greedy extraction strategy that has been shown to be effective for certain domains [PSSWT15, WHL$^+$20, WWF$^+$20]. For each e-class, the greedy strategy computes the total cost of the subtrees rooted on each of the e-nodes, and picks the e-node with the smallest subtree cost.

Greedy extraction is not guaranteed to extract the graph with the minimum cost, even under our independent cost model. For example, if two children of an e-node share a subgraph, greedy extraction would ignore the sharing and overestimate the cost.

ILP extraction The second approach we experiment with is formulating the extraction problem as an Integer Linear Program (ILP).

Let $i = 0, ..., N - 1$ be the set of e-nodes in the e-graph. Let $m = 0, ..., M - 1$ be the set of e-classes in the e-graph. Let e_m denote the set of e-nodes within e-class m: $\{i | i \in e_m\}$. Let h_i denote the set of children e-classes for e-node i. Let $g(i)$ denote the e-class of e-node i, i.e. $i \in e_{g(i)}$. Let $m = 0$ be the root e-class. Each e-node is associated with a cost c_i.

We then formulate our problem as follows:

$$\text{Minimize: } f(x) = \sum_i c_i x_i$$

Subject to:

$$x_i \in \{0, 1\}, \tag{6.1}$$

$$\sum_{i \in e_0} x_i = 1, \tag{6.2}$$

$$\forall i, \forall m \in h_i, x_i \leq \sum_{j \in e_m} x_j, \tag{6.3}$$

$$\forall i, \forall m \in h_i, t_{g(i)} - t_m - \epsilon + A(1 - x_i) \geq 0, \tag{6.4}$$

$$\forall m, 0 \leq t_m \leq 1, \tag{6.5}$$

Here we introduce a binary integer variable x_i for each e-node i; node i is selected if $x_i = 1$, and not selected otherwise. Constraint (2) ensures that one node is picked in the root e-class. Constraint (3) ensures that if a node is picked, then at least one node in each of its children e-classes needs to be picked. We rely on the fact that at the optimal solution, each e-class can have at most one picked node (otherwise we can remove more picked nodes in this e-class to reduce the objective while still satisfying all the constraints). Constraints (1)–(3) and the objective encode the main extraction logic.

A more subtle requirement on the extraction phase is that the extracted graph cannot contain cycles. While the e-graph can (and likely will) contain cycles, the extracted graph is meant to map directly to an executable tensor DAG. The extraction procedure must therefore take care to respect the acyclic invariant of DAGs.

?? shows an example to illustrate how valid rewrites can produce cycles in the e-graph. To ensure the extracted graph does not contain cycles, we introduce a real variable t_m for each e-class m in the ILP. Constraint (4) ensures that the order defined by t_m's is a valid topological order for the extracted graph. Here $\epsilon < 1/M$ is a small constant for effectively encoding strict inequalities in ILP. A is a large enough constant such that $A > 1 + \epsilon$. Constraint (5) is to limit the range for the topological order variables t_m's.

We also experiment with using integer variables for t_m's. In this case, t_m's are constrained to take integer values between 0 to $M - 1$. Constraint (4) changes accordingly to: $\forall i, \forall m \in h_i, t_{g(i)} - t_m + A(1 - x_i) \geq 1$, where $A \geq M$.

Unlike greedy extraction, the optimal solution to the ILP is guaranteed to give a valid graph (no cycles) with the lowest cost.

Cycle Filtering

Similar to previous work that uses ILP extraction [TSTL09, WHL$^+$20], we find that as the size of the e-graph grows bigger, the ILP solver takes a long time and becomes the main bottleneck. This is mainly due to the cycle constraint (4): ILP solver struggles to find a feasible solution with these constraints. Therefore, we explore an alternative approach by filtering cycles during the exploration phase to make sure that the e-graph does not contain any cycles at the end of the exploration phase. This way, we can get rid of the cycle constraints in the ILP.

Vanilla cycle filtering The first method is to check if applying a substitution introduces cycles to the e-graph, and discard such a substitution. This check is run every time before applying a substitution. Each check requires a pass over the entire e-graph. For one iteration during the exploration phase, if we denote N as the current size of the e-graph and n_m as the total number of matches of the rewrite rules on the e-graph, then this vanilla cycle filtering has complexity $\mathcal{O}(n_m N)$.

Efficient cycle filtering As the number of matches n_m is typically large and scales with N, vanilla cycle filtering can be slow. We therefore design a novel and more efficient cycle filtering algorithm, consisting of a *pre-filtering* step and a *post-processing* step. Algorithm 2 shows the pseudocode for the exploration phase with efficient cycle filtering.

At the start of each iteration, we do one pass over the e-graph to record the set of descendent e-classes for each e-node (stored in a descendants map). During the iteration, for each match of

Algorithm 2 Exploration phase with efficient cycle filtering

```
Input:  starting e-graph $\mathcal{G}$, set of rewrite rules $\mathcal{R}$.
Output:  updated e-graph $\mathcal{G}$, filter list $l$
 1: $l = \{\}$
 2: for iter = 0, …, MAX_ITER do
 3:     descendants map $d$ = GetDescendants($\mathcal{G}, l$)
 4:     matches = Search($\mathcal{G}, \mathcal{R}, l$)
 5:     for match $\in$ matches do
 6:         if not WillCreateCycle(match, $d$) then
 7:             Apply($\mathcal{G}$, match)
 8:         end if
 9:     end for
10:     while true do
11:         cycles = DfsGetCycles($\mathcal{G}, l$)
12:         if len(cycles) == 0 then
13:             break
14:         end if
15:         for cycle $\in$ cycles do
16:             ResolveCycle($\mathcal{G}, l$, cycle)
17:         end for
18:     end while
19: end for
20: return $\mathcal{G}, l$
```

the rewrite rules, we use the pre-stored descendants map to check if applying a rewrite introduces cycles to the e-graph; if so, we skip this match. Line 3–9 implements the pre-filtering step. Notice that this check is sound but not complete: a match that passes this check can still introduce cycles to the e-graph. This is because new descendants relations introduced by the previous rewrite in this iteration are not included in the pre-stored descendants map.

To resolve the cycles we missed in the pre-filtering step, we add a post-processing step at the end of each iteration (line 10-18). We make a pass over the e-graph in DFS order and collect a set of cycles in the e-graph. For each cycle, we choose the last node that is added to the e-graph, and add that node to a filter list. The nodes in the filter list are considered as removed from the e-graph. We make sure those nodes are not picked during extraction by explicitly adding constraints $\forall i \in l, x_i = 0$ to the ILP.

By constructing a descendants map once before each iteration, each of the checking in the pre-filtering step takes constant time. The worst case complexity of the post-processing step is $\mathcal{O}(n_c N)$, where n_c is the number of cycles in the e-graph. Since n_c is typically much smaller than n_m, this algorithm is much faster than the vanilla cycle filtering. In practice each DFS pass over the e-graph can find many cycles, which makes $\mathcal{O}(n_c N)$ a very conservative upper bound.

6.4.4 Evaluation

We implement Tensat in Rust [Rus] using `egg` [WWF+20], an open source equality saturation library. For the extraction phase, we use SCIP [GAB+20] as the ILP solver, wrapped by Google OR-tools [PF].

We utilize egg's *e-class analysis* feature for the shape checking discussed in Section 6.4.2. An e-class analysis associates data with each e-class to support rewrites that are not purely syntactic. We store all the relevant information of the tensors (shape, layout, split locations) in the analysis data and use these information for shape checking.

Experimental Setup

We compare Tensat with TASO [JPT+19] to evaluate our equality saturation based search. We use the same set of rewrite rules as TASO for our experiments. We evaluate on the inference graphs of 7 models: **BERT** [DCLT19], **ResNeXt-50** [XGD+17], **NasNet-A** [ZVSL18], **NasRNN** [ZL17], **Inception-v3** [SVI+16], **VGG-19** [LD15], and **SqueezeNet** [IMA+17]. This benchmark set covers a wide range of commonly used state-of-the-art models, including both models for computer vision tasks and models for NLP tasks, both human-designed models and automatically-discovered models by neural architecture search. We perform all experiments on a Google Cloud instance with one NVIDIA Tesla T4 GPU, a 16-core CPU, and 60 GB of memory. We also experiment with ResNet-50 [HZRS16], but find that on T4 GPU, the rewrite rules from TASO cannot provide any speedup to the graph.

For Tensat, our full approach uses the efficient cycle filtering algorithm (Section 6.4.3) during the exploration phase and the ILP method without the cycle constraints (Section 6.4.3) for extraction. We set a limit on the number of nodes in the e-graph $N_{\text{max}} = 50000$ and the number of iterations for exploration $k_{\text{max}} = 15$. We terminate the exploration phase when any of the limit is reached, or the e-graph is saturated. We set a separate limit k_{multi} on the number of iterations to apply the multi-pattern rules. We use a default of $k_{\text{multi}} = 1$ for the main results in Section 6.4.4 and Section 6.4.4, and study the effect of varying k_{multi} in **??**. We set a timeout of 1 hour for the ILP solver.

For TASO's backtracking search, we use their default settings from their artifact evaluation code on the number of iterations[2] $n = 100$ and the hyperparameter $\alpha = 1.0$ for each benchmark. We also test $\alpha = 1.05$ as mentioned in their paper, and find that the difference is tiny (difference in speedup percentage is less than 0.1% on average over the benchmarks). Increasing to $n = 1000$ leads to less than 1% speedup gain with the cost of over 11x longer in optimization time on average.

Program Speedup

We compare the speedup percentage of the optimized graph with respect to the original graph between Tensat and TASO. We use TASO's cuDNN backend to measure the runtime of the full computation graphs. Figure 6.3 shows the results. We can see that Tensat discovers better optimized graphs compared with TASO's backtracking search in most benchmarks. Tensat's optimized graphs are on average 6.6% faster than TASO's. We see the biggest speedup of 23% over TASO on NasRNN. Note that for Inception-v3, Tensat with $k_{\text{multi}} = 1$ gives a smaller speedup than TASO, but increasing k_{multi} to 2 achieves a better speedup than TASO while still being 13.4× faster than TASO's search (see Figure 6.4).

This improvement comes from the fact that equality saturation covers a much larger space of equivalent graphs than sequential backtracking search. By using e-graph as a compact representation of an exponential number of equivalent graphs, Tensat is able to cover orders of magnitude

[2]The number of iterations of the outer loop, see Algorithm 2 in [JPT^{+}19] for more details

more equivalent graphs than TASO.

We inspect the optimized graphs from Tensat and recorded some rewrite patterns that is used in them. We present several examples of useful patterns in the Appendix.

Optimization Time

Another important metric is the time taken by the optimizer itself. For Tensat, this is the sum of time taken by the exploration phase and the extraction phase. For TASO, we record two times for a single backtracking search. The first is the total time of the backtracking search with the default number of iterations (T_{total}). The second one is the time taken to first reach the best graph found during its search (T_{best}). T_{best} is the best possible time for TASO's sequential backtracking search. In practice, it is difficult (if not impossible) to achieve T_{best} since the sequential search algorithm would have no way to know that it can stop at that point.

Figure 6.4 shows the time taken by the optimizers across benchmarks. We can see that Tensat runs 9.5x to 379x faster than TASO's T_{total}, and 1.8x to 260x times faster than T_{best}. This shows that Tensat can not only cover a much larger search space, but also achieve this in drastically less time. Furthermore, Tensat's optimization time is small enough that we believe our approach can be integrated into a default compilation flow instead of running the search as an additional offline autotuning process.

Ablation Study

In this section, we study the effect of the important design choices in our approach.

Greedy v.s. ILP extraction The first important design choice is the extraction method. Table 6.3 shows the comparison between greedy extraction and ILP extraction. Although greedy extraction works fine on some benchmarks (e.g. NasRNN), it fails to extract an optimized graph on others (e.g. BERT and NasNet-A). This is due to the nature of greedy extraction: it makes the choices on which node to pick separately and greedily, without considering the inter-dependencies between the choices. Consider the rewrite in Figure 6.2 (merging two `matmuls` by `concat` and `split`) as an illustrative example. After applying this rewrite to the e-graph, there will be two e-classes that have multiple e-nodes: one e-class per each output. This rewrite can reduce the cost only if both e-classes choose the split node, since the RHS subgraph can be reused by the two outputs. However, greedy extraction will never pick the split nodes, since it does not know the RHS subgraph is shared between the two split nodes.

ILP with or without cycle constraints Here we study the effect of whether or not to include the cycle constraints in ILP. Table 6.4 presents the effect on extraction time as k_{multi} (thus e-graph size) varies. With the cycle constraints, ILP solver time quickly increases with the e-graph size, and reaches timeout when $k_{\text{multi}} = 2$. In our experiments, the ILP solver has not yet found a feasible solution at timeout. Removing the cycle constraints leads to approximately 10x–1000x speedup on ILP solving time on larger e-graphs. These results show that the main difficulty for the ILP solver is to satisfy the cycle constraints. Thus, removing the cycle constraints makes it possible for our approach to scale to larger e-graphs.

Efficient cycle filtering To remove the cycle constraints from ILP, we need to perform cycle filtering during the exploration phase. Here we compare the two cycle filtering techniques introduced in Section 6.4.3. Table 6.5 shows the effect on the exploration phase time, as k_{multi} varies. We can see that the efficient cycle filtering algorithm achieves up to 2000x speedup compared with the vanilla algorithm, making it possible to explore a larger e-graph.

Table 6.2: Operators supported by Tensat. There are four types for the nodes in our representation: tensor type (T), string type (S), integer type (N), and tensor tuple type (TT). The integer type is used to represent parameters of the operators, such as stride, axis, and also padding and activation modes (by representing different modes using different integers). The more complex, variable-length parameters (e.g. shape, axes permutation) are represented using the string type according to the specified formats.

Operator	Description	Inputs	Type signature
ewadd	Element-wise addition	input_1, input_2	$(T, T) \to T$
ewmul	Element-wise multiplication	input_1, input_2	$(T, T) \to T$
matmul	Matrix multiplication	activation, input_1, input_2	$(N, T, T) \to T$
conv [a]	Grouped convolution	stride_h, stride_w, pad., act., input, weight	$(N, N, N, N, T, T) \to$
relu	Relu activation	input	$T \to T$
tanh	Tanh activation	input	$T \to T$
sigmoid	Sigmoid activation	input	$T \to T$
poolmax	Max pooling	input, $\text{kernel}_{\{h,w\}}$, $\text{stride}_{\{h,w\}}$, pad., act.	(T, N, N, N, N, N, N)
poolavg	Average pooling	input, $\text{kernel}_{\{h,w\}}$, $\text{stride}_{\{h,w\}}$, pad., act.	(T, N, N, N, N, N, N)
transpose [b]	Transpose	input, permutation	$(T, S) \to T$
enlarge [c]	Pad a convolution kernel with zeros	input, ref-input	$(T, T) \to T$
concat_n [d]	Concatenate	axis, $\text{input}_1, \ldots, \text{input}_n$	$(N, T, \ldots, T) \to T$
split [e]	Split a tensor into two	axis, input	$(N, T) \to TT$
split_0	Get the first output from split	input	$TT \to T$
split_1	Get the second output from split	input	$TT \to T$
merge [f]	Update weight to merge grouped conv	weight, count	$(T, N) \to T$
reshape [g]	Reshape tensor	input, shape	$(T, S) \to T$
input	Input tensor	identifier [h]	$S \to T$
weight	Weight tensor	identifier [h]	$S \to T$
noop [i]	Combine the outputs of the graph	input_1, input_2	$(T, T) \to T$

[a] Same representation as TASO [JPT+19]. Normal and depth-wise convolutions are special cases of grouped convolutions.
[b] Axis permutation for transpose is specified using a string with format: axis_1_axis_2_....
[c] Pad a convolution kernel (input) with zeros to make it the same size as ref-input.
[d] Since each type of node needs to have a fixed number of inputs, we have a separate concat for each number of inputs.
[e] Split the tensor in the given axis. The position of the split is at the place of the most recent concat.
[f] Merge every *count* number of groups in the grouped convolution. See TASO [JPT+19] for more details.
[g] Specify the target shape using a string with format: dim_1_dim_2_....
[h] The identifier for an input or weight tensor contains its name and shape, specified as a string with format: name@dim_1_dim_2_....
[i] For combining the outputs of the graph to make the graph single-rooted. No actual operator is associated with noop.

figures/rewrite-example.pdf

Source: (matmul ?input$_1$?input$_2$), (matmul ?input$_1$?input$_3$)
Target: (split$_0$ (split 1 (matmul ?input$_1$ (concat$_2$ 1 ?input$_2$?input$_3$)))),

(split$_1$ (split 1 (matmul ?input$_1$ (concat$_2$ 1 ?input$_2$?input$_3$))))

Figure 6.2: Example rewrite rule and its representation in S-expressions. Identifiers starting with "?" denote variable nodes. For clarity, we omit the activation mode inputs to `matmul`. Arrows point from parent nodes to children nodes. 1 is the axis for `split` and `concat` operators.

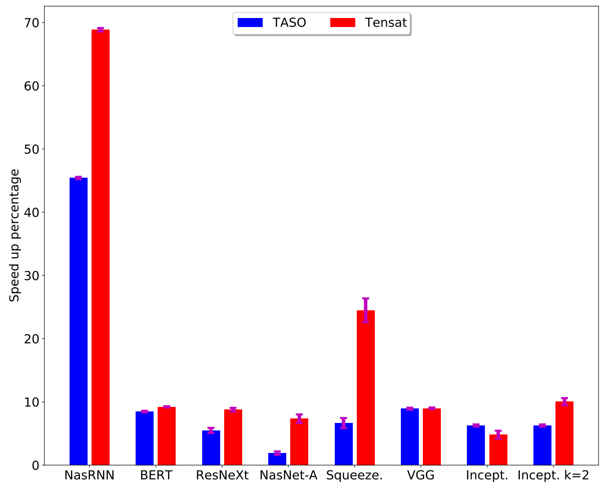


Figure 6.3: Speedup percentage of the optimized graph with respect to the original graph, TASO v.s. Tensat. Each setting (optimizer × benchmark) is run for five times, and we plot the mean and standard error for the measurements.

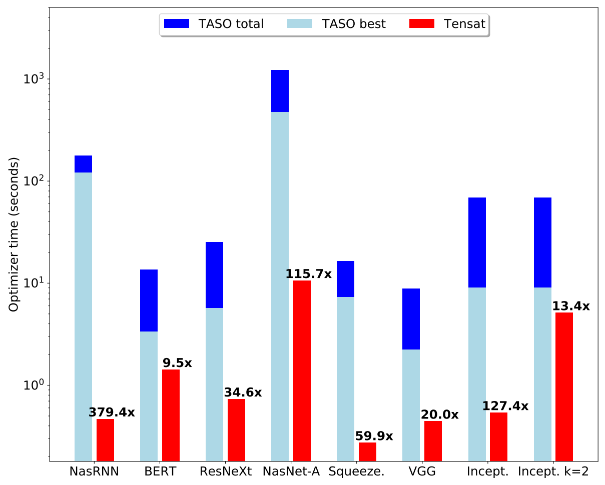


Figure 6.4: Comparison of the optimization time (log scale) between TASO and Tensat. "TASO total" is the total time of TASO search. "TASO best" indicates when TASO found its best result; achieving this time would require an oracle telling it when to stop.

Graph Runtime (ms)	**Original**	**Greedy**	**ILP**
BERT	1.88	1.88	**1.73**
NasRNN	1.85	1.15	**1.10**
NasNet-A	17.8	22.5	**16.6**

Table 6.3: Comparison between greedy extraction and ILP extraction, on BERT, NasRNN, and NasNet-A. This table shows the runtime of the original graphs and the optimized graphs by greedy extraction and ILP extraction. The exploration phase is run with $k_{\text{multi}} = 1$.

Extraction time (s)	$k_{\textbf{multi}}$	**With cycle** real	int	**Without cycle**
BERT	1	0.96	0.98	**0.16**
	2	>3600	>3600	**510.3**
NasRNN	1	1116	1137	**0.32**
	2	>3600	>3600	**356.7**
NasNet-A	1	424	438	**1.81**
	2	>3600	>3600	**75.1**

Table 6.4: Effect of whether or not to include cycle constraints in ILP on extraction time (in seconds), on BERT, NasRNN, and NasNet-A. For the cycle constraints, we compare both using real variables and using integer variables for the topological order variables t_m.

k_{multi}	BERT		NasRNN		NasNet-A	
	Van.	Eff.	Van.	Eff.	Van.	Eff.
1	0.18	**0.17**	1.30	**0.08**	3.76	**1.27**
2	32.9	**0.89**	2932	**1.47**	$>$3600	**8.62**

Table 6.5: Comparison between vanilla cycle filtering and efficient cycle filtering, on the exploration phase time (in seconds) for BERT, NasRNN, and NasNet-A.

Bibliography

[AEH$^+$99] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. *Sci. Comput. Program.*, 34(1):1–54, April 1999.

[DCLT19] J. Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*, 2019.

[Der93] Nachum Dershowitz. *A taste of rewrite systems*, pages 199–228. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.

[dMB07] Leonardo de Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In Frank Pfenning, editor, *Automated Deduction – CADE-21*, pages 183–198, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. 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*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. *J. ACM*, 52(3):365–473, May 2005.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[DST80] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, October 1980.

[FSWC20] Jingzhi Fang, Yanyan Shen, Yue Wang, and Lei Chen. Optimizing DNN Computation Graph Using Graph Substitutions. *Proc. VLDB Endow.*, 13(12):2734–2746, July 2020.

[GAB$^+$20] Gerald Gamrath, Daniel Anderson, Ksenia Bestuzheva, Wei-Kun Chen, Leon Eifler, Maxime Gasse, Patrick Gemander, Ambros Gleixner, Leona Gottwald, Katrin Halbig, Gregor Hendel, Christopher Hojny, Thorsten Koch, Pierre Le Bodic, Stephen J. Maher, Frederic Matter, Matthias Miltenberger, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Christine Tawfik, Stefan Vigerske, Fabian Wegscheider, Dieter Weninger, and Jakob Witzig. The SCIP Optimization Suite 7.0. Technical report, Optimization Online, March 2020.

[HZRS16] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[IMA$^+$17] Forrest N. Iandola, Matthew W. Moskewicz, K. Ashraf, Song Han, W. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *ArXiv*, abs/1602.07360, 2017.

[JNR02] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: A goal-directed superoptimizer. *SIGPLAN Not.*, 37(5):304–314, May 2002.

[JPT$^+$19] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.

[JTW+19] Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing DNN Computation with Relaxed Graph Substitutions. In *Proceedings of the 2nd SysML Conference*, SysML '19, 2019.

[LD15] S. Liu and W. Deng. Very deep convolutional neural network based image classification using small training sample size. In *2015 3rd IAPR Asian Conference on Pattern Recognition (ACPR)*, pages 730–734, 2015.

[Nel80] Charles Gregory Nelson. *Techniques for Program Verification*. PhD thesis, Stanford, CA, USA, 1980. AAI8011683.

[NO05] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In *Proceedings of the 16th International Conference on Term Rewriting and Applications*, RTA'05, page 453–468, Berlin, Heidelberg, 2005. Springer-Verlag.

[NWA+20] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. Synthesizing structured CAD models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 31–44, New York, NY, USA, 2020. Association for Computing Machinery.

[PF] Laurent Perron and Vincent Furnon. Or-tools.

[PKSL20] Varot Premtoon, James Koppel, and Armando Solar-Lezama. Semantic code search via equational reasoning. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 1066–1082, New York, NY, USA, 2020. Association for Computing Machinery.

[PSSWT15] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. *SIGPLAN Not.*, 50(6):1–11, June 2015.

[Rus] Rust. Rust programming language. `https://www.rust-lang.org/`.

[STL11] Michael Stepp, Ross Tate, and Sorin Lerner. Equality-based translation validator for llvm. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 737–742, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[SVI$^+$16] Christian Szegedy, V. Vanhoucke, S. Ioffe, Jon Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2818–2826, 2016.

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, April 1975.

[TSTL09] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 264–276, New York, NY, USA, 2009. ACM.

[WHL$^+$20] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. SPORES: Sum-product optimization via relational equality saturation for large scale linear algebra. *Proceedings of the VLDB Endowment*, 2020.

[WWF$^+$20] Max Willsey, Yisu Remy Wang, Oliver Flatt, Chandrakana Nandi, Pavel Panchekha, and Zachary Tatlock. egg: Fast and extensible e-graphs, 2020.

[WZN^{+}19] Chenming Wu, Haisen Zhao, Chandrakana Nandi, Jeffrey I. Lipton, Zachary Tatlock, and Adriana Schulz. Carpentry compiler. *ACM Transactions on Graphics*, 38(6):Article No. 195, 2019. presented at SIGGRAPH Asia 2019.

[XGD^{+}17] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He. Aggregated residual transformations for deep neural networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5987–5995, 2017.

[ZL17] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. 2017.

[ZVSL18] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8697–8710, 2018.