Efficient Extraction for Effectful E-graphs

OLIVER FLATT, University of Washington, USA
ANJALI PAL, University of Washington, USA
YIHONG ZHANG, University of Washington, USA
RYAN TJOA, University of Washington, USA
KIRSTEN GRAHAM, University of Washington, USA
ALEX FISCHMAN, University of Washington, USA
CHANDRAKANA NANDI, Certora Inc., USA and University of Washington, USA
ELI ROSENTHAL, Google, USA
ZACHARY TATLOCK, University of Washington, USA
HAOBIN NI, University of Washington, USA

E-graphs have enabled recent advances in program optimization, synthesis, and verification, yet remain difficult to apply to effectful programs whose memory and I/O operations must respect execution order. Existing effect-aware extraction algorithms rely on integer linear programming (ILP) and dominate total runtime.

We introduce **STATEWALK DP**, a new extraction algorithm that enforces effect ordering efficiently without external solvers. We prove that finding *any* effect-safe extraction is NP-complete, but show that STATEWALK DP is tractable in *statewalk width*, a parameter that measures the complexity of dataflow interactions among effects. In practice, statewalk width generally remains small, enabling STATEWALK DP to achieve order-of-magnitude speedups over ILP extraction while producing programs comparable to LLVM across our benchmarks. We implement the algorithm in EGGCC, a prototype e-graph-based compiler for imperative Bril programs, and demonstrate that effect-aware extraction is no longer a bottleneck.

CCS Concepts: • **Software and its engineering → Compilers**; *Automated static analysis.*

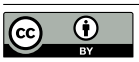
Additional Key Words and Phrases: equality saturation, e-graphs, extraction, effects, compiler optimization, term rewriting

ACM Reference Format:
Oliver Flatt, Anjali Pal, Yihong Zhang, Ryan Tjoa, Kirsten Graham, Alex Fischman, Chandrakana Nandi, Eli Rosenthal, Zachary Tatlock, and Haobin Ni. 2026. Efficient Extraction for Effectful E-graphs. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 398 (October 2026), 28 pages. https://doi.org/10.1145/3839530

1 Introduction

Equality saturation and e-graphs [15, 25, 43, 49] power new state-of-the-art program optimizers, synthesizers, and verifiers across a variety of domains [3, 5, 8, 14, 16, 24, 30, 32, 38, 40, 44, 47, 50].

Authors' Contact Information: Oliver Flatt, University of Washington, Seattle, Washington, USA, oflatt@cs.washington.edu; Anjali Pal, University of Washington, Seattle, Washington, USA, anjalip@cs.washington.edu; Yihong Zhang, University of Washington, Seattle, Washington, USA, yz489@cs.washington.edu; Ryan Tjoa, University of Washington, Seattle, Washington, USA, rtjoa@cs.washington.edu; Kirsten Graham, University of Washington, Seattle, Washington, USA, kmgraham@cs.washington.edu; Alex Fischman, University of Washington, Seattle, Washington, USA, adfisch@cs.washington.edu; Chandrakana Nandi, Certora Inc., Seattle, Washington, USA and University of Washington, Seattle, Washington, USA, cnandi@cs.washington.edu; Eli Rosenthal, Google, Oakland, California, USA, ezrosenthal@gmail.com; Zachary Tatlock, University of Washington, Seattle, Washington, USA, ztatlock@cs.washington.edu; Haobin Ni, University of Washington, Seattle, Washington, USA, hn42@cs.washington.edu.



ACM 2475-1421/2026/10-ART398
https://doi.org/10.1145/3839530

These tools follow a common workflow: initialize an e-graph $\mathcal{G}$ with input program P; expand $\mathcal{G}$ by applying semantics-preserving rules to build a large space of programs equivalent to P; then *extract* an optimized version of P from $\mathcal{G}$ guided by a cost function (e.g., program size). Decoupling the exploration of equivalent program fragments from extraction mitigates phase ordering [18], freeing engineers to focus on semantics-preserving optimization rules, not transformation order.

Most e-graph-based optimizers only operate over *pure* programs (i.e., those without side effects), though as we show in a case study, optimizing *effectful* programs using e-graphs can enable significant speedups compared to LLVM (Section 8.3). Rewriting effectful programs using e-graphs is difficult due to the tension between effectful optimizations and congruence. Congruence is a fundamental property of e-graphs that requires all equivalent terms to be interchangeable, which is essential for efficient representation and exploration of equivalent programs. Unfortunately, under congruence even basic rewrites over effectful programs lead to *ambiguous* terms whose subterms require incompatible effect orders. The only viable approach for effectful rewrites in an e-graph is to tolerate these ambiguous terms, thereby relaxing the equivalence relation represented by the e-graph. Optimization rules must still be semantics-preserving according to the relaxed relation, but are free to relate terms with equivalent effect orderings. This relaxed equivalence relation creates the key challenge hindering effectful e-graphs: enforcing valid effect ordering at extraction time, which we call *effect-safe extraction*.

Optimal extraction from pure e-graphs is known to be NP-complete [34, 48], but prior to this work there was no known complexity result characterizing effect-safe extraction[2]. We show that optimal effect-safe extraction is NP-complete, and surprisingly, that finding *any* effect-safe extraction is NP-complete as well (Section 5). Table 1 shows the complexity of pure and effect-safe extraction.

	Pure	Effect-Safe
Any	P	**NP-complete**
Optimal	NP-complete	**NP-complete**

Table 1. Complexity bounds of extraction. Even without considering effect ordering, finding the optimal extraction with respect to a cost model is known to be NP-complete [34, 48]. Bold indicates our contribution.

	Pure	Effect-Safe
Best-Effort[1]	Many, see §9	[38], **STATEWALK DP**
Optimal	[11]	Future work

Table 2. Practical approaches to extraction. Most applications of pure e-graphs use greedy best-effort extraction algorithms, which work well in practice [4, 27, 42, 46]. Bold indicates our contribution.

Of course, just because a problem is NP-complete does not mean it goes away. In practice, researchers have developed many extraction algorithms for pure e-graphs, summarized in Table 2. A few systems sidestep the challenge of effect-safe extraction: they either give up on effectful terms and limit e-graphs to pure program fragments only [8, 15, 22], or leave soundness entirely to the user [47]. Both approaches fall in the pure (top-left) quadrant of Table 2 because they do not enforce effect ordering constraints during extraction. The only sound prior approach, introduced by Peggy [38] (top-right), requires calling out to an integer linear programming (ILP) solver to enforce effect ordering constraints. However, Peggy's ILP encoding is incomplete, so it is a best-effort extraction algorithm[2]. Furthermore, relying on a general-purpose ILP solver leads to unpredictable performance and frequent failures, as seen in both prior work and our own reimplementation (Figure 1). Even when ILP succeeds, extraction typically dominates total optimization time, a problem we call the *effectful e-graph extraction bottleneck*.

In this paper, we present the first practical effect-safe extraction algorithm, STATEWALK DP, which eliminates the bottleneck for best-effort approaches. It is guaranteed to produce an effect-safe

[1]Note that in the context of optimization, the input program is a trivial, yet unsatisfying, solution to the extraction problem. Instead of attempting to find *any* extraction, most implementations take a greedy approach to produce best-effort results.
[2]Peggy [38] uses an ILP solver to tackle the effect-safe extraction problem, but its encoding is incomplete, so it does not imply any complexity bound on the effect-safe extraction problem despite ILP being NP-complete. From Tate et al. [38]: "We have developed a simple constraint solving technique for finding a linearization of heap operations in a PEG, but this technique is not complete (as in, even if there is a linearization, we are not guaranteed to find it)."

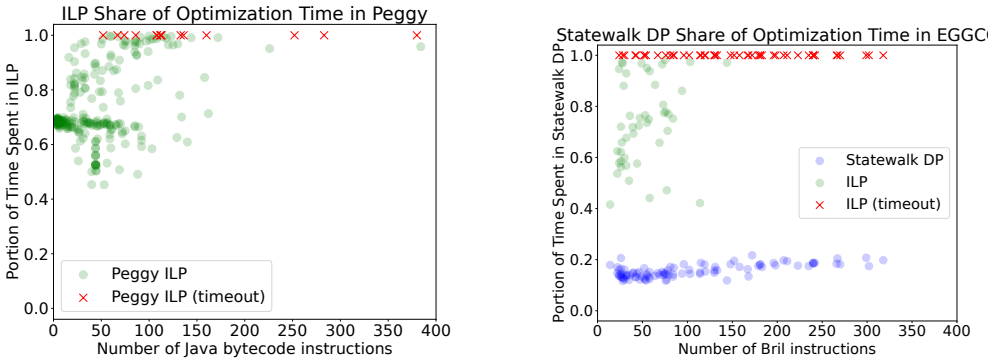


Fig. 1. *The effectful e-graph extraction bottleneck.* For ILP-based approaches (green), the majority of total optimization time is often spent on extraction, both in prior work [38] (left) and our EGGCC prototype (right). Although these graphs differ in source languages, benchmarks, and implementations, both show the bottleneck and the unpredictability of existing approaches. In contrast, STATEWALK DP (blue) behaves predictably and significantly reduces extraction time. Section 8 details the setup for both plots.

extraction while optimizing for cost greedily. Our key idea is to *refine* the relaxed equivalence used by rules into an effect-safe version during extraction. STATEWALK DP partitions terms by equivalence classes over their *statewalks*: sequences of effectful operations along which their subterms may be reached. STATEWALK DP then uses a cost function and dynamic programming to select a single effect-safe extraction for each partition. We call the maximum number of partitions for any equivalence class the e-graph's *statewalk width*. We show that effect-safe extraction is fixed-parameter tractable (FPT) [7] in the statewalk width parameter: STATEWALK DP takes $O(kn^2)$ time where k is the statewalk width and n is the size of the e-graph, extending the popular $O(n^2)$ greedy algorithm to effectful e-graphs.

To evaluate STATEWALK DP, we developed EGGCC, a proof-of-concept optimizer. EGGCC translates Bril programs [33] to an RVSDG-based representation [31], performs equality saturation using EGGLOG [49], extracts with STATEWALK DP, and finally translates the optimized program back to Bril. We found that statewalk width generally remains small in practice (mean of 33.15), enabling STATEWALK DP to achieve a 520× speedup over ILP-based extraction while producing code comparable to LLVM across our benchmarks.

This paper makes the following contributions:

- We identify the effectful e-graph extraction bottleneck and prove that both optimal extraction and *any* extraction in the face of effects are NP-complete in general (Section 5).
- We present STATEWALK DP, a new effect-aware extraction algorithm that is tractable with respect to statewalk width and produces best-effort results with respect to the cost model. We prove STATEWALK DP is sound and complete, and analyze its time complexity (Section 6).
- We implement STATEWALK DP in EGGCC, a proof-of-concept e-graph-based optimizer for imperative Bril [33] programs (Section 7). Through a case study, we show the value of e-graphs for optimizing effectful programs: adding a single domain-specific optimization enables significant speedups without interfering with existing ones (Section 8.3).
- We evaluate STATEWALK DP's extraction speed and the quality of the programs it extracts, demonstrating significant speedups over prior ILP-based approaches while producing high-quality extractions (Section 8).

2 Background

We provide background on e-graphs and dataflow IRs that represent effectful programs.

2.1 Formal Definition of an E-graph

This section gives a formal definition for e-graphs which is built upon in Section 4. We start by defining terms, then the congruence closure, and finally the e-graph as a data structure representing the congruence closure over a set of identities. Note that while some recent work defines e-graphs as graphs with nodes and edges [43], our definition leverages the more classic definition of a congruence closure [6]. The two definitions are equivalent, with terms taking the place of e-nodes. We use the classic definition because it eases formalization and proofs.

Given a set of ranked function symbols Σ, we inductively define terms of Σ as $t = f(t_1, \ldots, t_n)$, where f is an n-ary function symbol and $t_1, \ldots t_n$ are terms. We call the finite sequence $\langle t_1, \ldots, t_n \rangle$ the *children* of t and the function symbol f the *head* of t. We denote the children of a term t with $\mathsf{Children}(t)$. Constants, such as integers, are represented as nullary function symbols and are written as f instead of $f()$.

We use partial equivalences and partial congruences [17, 35] to distinguish terms represented by an e-graph from those that are not. A partial equivalence relation (PER) $\cong$ is a symmetric and transitive binary relation. A partial congruence relation is a PER augmented with the congruence and subterm properties:

$$f(t_1, \ldots, t_k) \cong f(t_1, \ldots, t_k) \wedge \bigwedge_{i=1\ldots k} t_i \cong t_i' \implies f(t_1, \ldots, t_k) \cong f(t_1', \ldots, t_k') \quad \text{(Congruence)}$$

$$f(t_1, \ldots, t_k) \cong f(t_1, \ldots, t_k) \implies \bigwedge_{i=1\ldots k} t_i \cong t_i \quad \text{(Subterm)}$$

Given a set of grounded identities $E = \{t_1 = t_2, \ldots, t_{m-1} = t_m\}$, the congruence closure of E is the smallest partial congruence relation containing E. Note that the congruence closure may be infinite even if E is finite.

E-graphs. An e-graph $\mathcal{G}$ is a data structure computed from grounded equalities E that compactly represents the congruence closure of E [6, 35]. We write the PER represented by an e-graph $\mathcal{G}$ as $\cong_{\mathcal{G}}$. Note that PERs are not necessarily reflexive: $x \cong_{\mathcal{G}} x$ holds for all x represented by the e-graph but it does not hold for terms that are not represented by the e-graph. An e-graph is a set of *e-classes*, and each e-class of an e-graph is a finite set of terms. Each e-class stores *representative terms* drawn from the subterms of the original set of grounded identities E. The e-classes of an e-graph have a one-to-one correspondence with the equivalence classes of the congruence closure. We say a term t is *represented* by e-class $C \in \mathcal{G}$ when $\exists t' \in C$ such that $t' \cong_{\mathcal{G}} t$. We denote the set of all terms represented by an e-class with $\mathbb{T}_{\mathcal{G}}(C)$. Given a term $t \in \mathbb{T}_{\mathcal{G}}(C)$, we define $\mathbb{C}_{\mathcal{G}}(t) = C$[2]. Note that $\mathbb{T}_{\mathcal{G}}(C)$ may be infinite, even though C is finite[3].

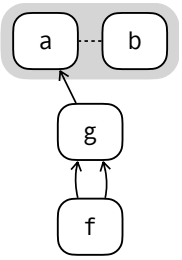


Fig. 2. An example e-graph.

Figure 2 shows an example e-graph. Dotted edges show equivalences and gray shading is used to highlight equivalence classes. It has three e-classes, with representative terms $\{a, b\}$, $\{g(a)\}$, and $\{f(g(a), g(a))\}$ respectively. The top e-class represents equivalences $a \cong_{\mathcal{G}} b$, the middle

[2]The representative terms in our definitions serve as e-nodes in frameworks like egg. Using concrete representative terms makes defining our algorithm more natural.

[3]This is traditionally explained by cycles in the e-graph between e-nodes. We use grounded terms in this formalization, so the cycles are formed by the combination of equivalences between terms and the parent-child relationship on terms.

e-class represents equivalences $g(a) \cong_{\mathcal{G}} g(b)$, and the bottom e-class represents equivalences $f(g(a), g(a)) \cong_{\mathcal{G}} f(g(a), g(b)) \cong_{\mathcal{G}} f(g(b), g(a)) \cong_{\mathcal{G}} f(g(b), g(b))$.

While the mathematical definition of an e-graph treats terms as trees, practical implementations store and visualize them as directed acyclic graphs (DAGs). Each node is annotated with a function symbol and has ordered outgoing edges. A term can be represented as a DAG by sharing common subexpressions, and a DAG can be unfolded into a tree by duplicating shared subterms. Implementations typically rely on hash-consing to ensure any identical term corresponds to exactly one node in the DAG [43]. For example, the term $f(g(a), g(a))$ in Figure 2 is represented as a DAG in which the subterm $g(a)$ is shared.

In the context of program optimization, declarative optimization rules are applied repeatedly over the entire e-graph to find new identities, which are then added to E. This technique is known as *equality saturation* [38] and is implemented in multiple modern frameworks [43, 49]. After rule application, an optimal program is *extracted* from the grown e-graph. More formally, an extraction of an e-class C of an e-graph $\mathcal{G}$ is a term $t \in \mathbb{T}_{\mathcal{G}}(C)$. The e-graph extraction problem is to find an extraction with low cost according to a cost function. This paper focuses on this extraction procedure, but for e-graphs containing effectful programs, i.e., **effectful e-graphs**.

2.2 Dataflow-based IRs and RVSDGs

E-graphs support a class of intermediate representations (IRs) based on dataflow graphs. Each node in a dataflow graph represents a value. Unlike control-flow graphs (CFGs), dataflow graphs directly encode data dependencies. Dataflow graphs enable substitution of equivalent nodes and therefore satisfy congruence, a key property of e-graphs.

In a dataflow-based IR, evaluating nodes in any topological order must yield the same value. Effectful programs, however, require ordering of effects. For example, when printing to stdout, the order of operations matters. Dataflow graph IRs support effectful programs by defining a special *state value* that is threaded through all effectful operators to enforce a total order on effects. All non-state values are *pure values*. Operators that produce state values are *effectful* and those that produce pure values are *pure*. To ensure the dataflow graph is *effect-safe*, state values must be used linearly: each effectful operation must consume a state value and produce a new one. In other words, the state value must not be duplicated or dropped. We call the unique sequence of effectful operations on the state value the *statewalk*.

Many dataflow IRs support control flow. Our implementation (Section 7) uses Regionalized Value State Dependence Graphs (RVSDGs) [1, 31].

3 Overview

This section illustrates the challenge of effect-safe extraction and outlines STATEWALK DP.

3.1 Effect-Safe Extraction

The core challenge is that not all terms represented by an effectful e-graph correspond to valid effect orderings. We call such terms *effect-unsafe*. Crucially, despite having a well-defined semantics (Section 4), effect-unsafe terms cannot be translated back to sequential programs (for code generation). At the same time, having effect-unsafe terms in an e-graph is unavoidable. This follows from congruence, a key invariant of e-graphs stating that equivalent subterms can be substituted for each other in any context. When the two equivalent subterms are effectful, this immediately leads to effect-unsafe terms with duplicated or dropped effects.

Figure 3 illustrates the issue. Panel I encodes a simple imperative program as a dataflow graph. The program contains four `Update` operations: C_1, C_2, C_3, C_4 (in order of execution). Each `Update` takes an address, a pure value, and a state value as arguments; writes the value to that address;

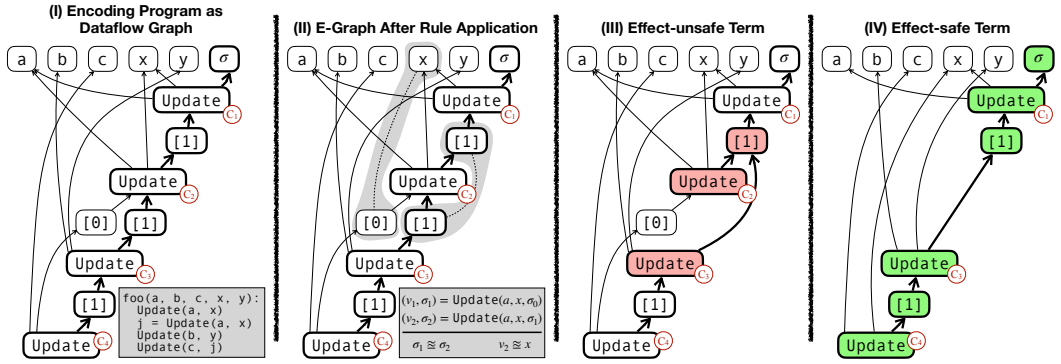


Fig. 3. (I) An imperative program encoded as a dataflow graph; (II) The e-graph after applying an optimization rule that eliminates a redundant `Update`; (III) An effect-unsafe term represented in the e-graph with two `Update` nodes relying on the same state value; (IV) An effect-safe term represented in the e-graph that uses the state value linearly. Data flows top to bottom: operators point to their operands. Here and in later figures, effectful operators are drawn with a bold outline; all other operators are pure.

and returns both the *old* value previously stored at that address (indexed as `[0]` in the figure), and an updated state value (indexed as `[1]`). Each effectful operation takes and returns a state value, which is threaded through the dataflow graph to encode the order of effects.

Panel II shows the e-graph after applying an optimization rule that merges redundant `Update` operations: when the same `Update` is repeated, the second performs no observable effect; it returns the same value and state as the first. The two ground equalities in the e-graph reflect this equivalence: the state after C_1 is equal to the state after C_2, and the pure value returned by C_2 is equal to `x`. However, the resulting e-graph now contains both effect-safe and effect-unsafe terms: congruence allows substituting a subterm with an equivalent one, even when both are effectful.

Panel III shows an effect-unsafe term represented in the e-graph. It is effect-unsafe because C_2 and C_3 share the same input state. Although this term has well-defined semantics (Section 4), it cannot be translated back to a valid program because the dataflow graph no longer defines a unique total order on effects. As a result, the effectful operations could be scheduled in different ways, producing different behaviors when memory addresses alias:

```
Update(a, x)                    Update(a, x)
j = Update(a, x)      - or -    Update(b, y)
Update(b, y)                    j = Update(a, x)
Update(c, j)                    Update(c, j)
```

Panel IV shows an effect-safe alternative. Each effectful operation consumes the state value produced by the previous one, establishing a single well-defined order.

The key difficulty is that effect-safety *does not compose under congruence*. A term may become effect-unsafe if one of its (effect-safe) subterms is substituted with another (effect-safe) term. In the example, substituting the pure result of C_2 for C_4's argument transforms the effect-safe term from Panel IV into the effect-unsafe one from Panel III.

To summarize, our goal is to efficiently find an effect-safe extraction—a term from the e-graph whose effectful operations form a single linear order—with a small cost. This is challenging because effect-safety does not compose. Section 5 shows that finding *any* effect-safe extraction is NP-complete in general. In practice, however, e-graphs produced by real program optimizers have additional structure, which StatewalkDP exploits to efficiently extract effect-safe terms.

3.2 Statewalks and Statewalk Width

The key idea behind STATEWALK DP is that different statewalks (sequences of effectful operations) can expose the same opportunities for extracting pure subterms. Rather than considering each statewalk independently, STATEWALK DP identifies classes of statewalks that enable the same extractions. This lets STATEWALK DP explore each class of equivalent (i.e., substitutable under congruence) statewalks once for each e-class, without enumerating the full, potentially infinite, space of possible statewalks.

Concretely, STATEWALK DP groups statewalks by the set of pure e-classes they make available. We call two statewalks *extractable set equivalent* when they are in the same e-class and enable the same set of extractable pure e-classes.

In Figure 3's running example, there are infinitely many statewalks to the root (bottom) e-class. Each statewalk may Update `a` any number of times before updating `b` and `c`. However, all these statewalks enable the same pure subterms. They are therefore extractable set equivalent, and an extractor is free to choose any of them; Appendix E shows the extractable sets STATEWALK DP computes for a small e-graph. STATEWALK DP keeps only the cheapest representative, which in this case yields the effect-safe extraction that avoids the redundant `Update` (Panel IV).

To efficiently exploit extractable set equivalence, STATEWALK DP maintains a dynamic programming (DP) table indexed by (e-class, extractable set) pairs. For each pair, the table stores the cheapest representative statewalk and the corresponding extracted term. This is sound because *extractable set equivalent statewalks are substitutable*: each leads to the same pure subterms and can be used interchangeably when building effect-safe terms. Starting from the leaves, STATEWALK DP propagates representative statewalks from child operands to their parent operators. This DP formulation reuses the shared decisions between extractable set equivalent statewalks.

We characterize the complexity of STATEWALK DP using a new parameter, *statewalk width*. The statewalk width of an e-class is the number of distinct extractable sets enabled by its statewalks, and the statewalk width k of an entire e-graph is the maximum statewalk width over its e-classes. In Figure 3, every statewalk exposes the same set of pure e-classes, so the statewalk width is 1. If an e-graph representing n terms has statewalk width k, STATEWALK DP runs in $O(kn^2)$ time. Our evaluation (Section 8) shows that statewalk width generally remains small in practice, enabling STATEWALK DP to extract effectful programs efficiently without sacrificing extraction quality.

The remainder of this paper formalizes the equivalences used in STATEWALK DP, defines statewalk width precisely, gives the full DP algorithm, and proves its correctness. We then evaluate STATEWALK DP's performance and extraction quality across a suite of benchmarks.

4 The Effect-Safe Extraction Problem

In this section, we formally define the effect-safe extraction problem. We first define the equivalence relation $\cong$ over a semantics of effectful dataflow IR including effect-unsafe terms. We formally define effect-safety and PER $\equiv_{\mathcal{G}}$, which is $\cong$ restricted to effect-safe programs in e-graph $\mathcal{G}$. Finally, we define the effect-safe extraction problem as finding an effect-safe term in an equivalence class of $\equiv_{\mathcal{G}}$ that corresponds to a root e-class of $\mathcal{G}$. Auxiliary definitions are collected in Appendix A.

4.1 Abstract Dataflow Language and $\cong$

We define (acyclic) dataflow graphs as terms as described in Section 2, making them amenable for formal treatment in an e-graph. Figure 4a gives the syntax of a minimal dataflow IR. We categorize values into pure ones and stateful ones. A pure value is a base value or a tuple of base values, and a stateful value is either a state or a tuple whose last value is a state. Tuples do not nest. A pure operator maps pure values to a pure value. An effectful operator takes any number of pure values

Base values $\phi \in \Phi = \{\bot, \top, \ldots\}$ States $\sigma \in \Sigma$ Pure values $v_p \in V^p ::= \phi \mid (\phi, \ldots, \phi)$

Stateful values $v_e \in V^e ::= \sigma \mid (\phi, \ldots, \phi, \sigma)$ Values $v ::= v_p \mid v_e$

Pure Operators $\mathtt{f}, \mathtt{g}, \ldots \in (V^p)^n \to V^p$ Effectful Operators $\mathtt{F}, \mathtt{G}, \ldots \in (V^p)^{n-1} \times V^e \to V^e$

Indices $\mathtt{n} \in \mathbb{N}$ Expression $\mathtt{e} ::= v \mid \mathtt{Arg} \mid \mathtt{f}(e_1, \ldots, e_n) \mid \mathtt{F}(e_1, \ldots, e_n) \mid (e, \ldots) \mid \mathtt{e}[n]$

(a) Syntax.

$$\frac{}{a \vdash v \Downarrow v} \qquad \frac{}{a \vdash \mathtt{Arg} \Downarrow a} \qquad \frac{a \vdash e_i \Downarrow v_i \quad \text{for } i = 0, \ldots, k-1}{a \vdash (e_0, \ldots, e_{k-1}) \Downarrow (v_0, \ldots, v_{k-1})} \qquad \frac{a \vdash e \Downarrow (v_0, \ldots, v_{n-1}) \qquad 0 \le k < n}{a \vdash e[k] \Downarrow v_k}$$

$$\frac{a \vdash e_i \Downarrow v_i \text{ for } i = 1, \ldots, n \qquad [\![\mathtt{f}]\!](v_1, \ldots, v_n) = v'}{a \vdash \mathtt{f}(e_1, \ldots, e_n) \Downarrow v'} \qquad \frac{a \vdash e_i \Downarrow v_i \text{ for } i = 1, \ldots, n \qquad [\![\mathtt{F}]\!](v_1, \ldots, v_n) = v'}{a \vdash \mathtt{F}(e_1, \ldots, e_n) \Downarrow v'}$$

(b) Big-step operational semantics.

Fig. 4. Syntax and semantics of a minimal effectful dataflow IR.

Equivalence	Description	Definition
$t_1 \cong t_2$	Equiv. under the semantics (Section 4.1)	$\forall a, v.\ a \vdash t_1 \Downarrow v \iff a \vdash t_2 \Downarrow v$
$t_1 \cong_{\mathcal{G}} t_2$	Equiv. in an effectful e-graph (Section 4.1)	$\exists C.\ t_1, t_2 \in \mathbb{T}_{\mathcal{G}}(C)$
$t_1 \equiv_{\mathcal{G}} t_2$	Effect-safe refinement of $\cong_{\mathcal{G}}$ (Section 4.2)	$t_1 \cong_{\mathcal{G}} t_2 \wedge \mathrm{EffSafe}(t_1) \wedge \mathrm{EffSafe}(t_2)$
$t_1 \equiv_{\mathrm{SW}} t_2$	Statewalk refinement of $\equiv_{\mathcal{G}}$ (Section 6.1)	$t_1 \equiv_{\mathcal{G}} t_2 \wedge \mathrm{effectful}(t_1) \wedge$ $\mathrm{effectful}(t_2) \wedge \mathrm{es}(t_1) = \mathrm{es}(t_2)$

Table 3. Notions of equivalence used in this paper, each introduced in the section named beside it, and each a refinement of the one above it. $a \vdash t \Downarrow v$ is the big-step judgement of Figure 4b. $\mathbb{T}_{\mathcal{G}}(C)$ is the set of terms represented by e-class C, and $\mathbb{C}_{\mathcal{G}}(t)$ is the e-class of a term t (Section 2.1). $\mathrm{EffSafe}(t)$ holds when t uses the state linearly (Definition 4.2), and $\mathrm{effectful}(t)$ when t evaluates to a stateful value (Section 4.2). $\mathrm{es}(t)$ is the extractable set of t: the pure e-classes that become extractable once the effects along t have been performed (Section 6.1).

plus a single state value, written last, and produces a stateful value. Every term therefore has at most one effectful child, so a statewalk (Section 4.2) follows a chain of effects unambiguously. An expression is a value, an `Arg` variable that refers to program inputs[4], an application of a pure or an effectful operator, a tuple of expressions, or an indexing into an expression.

Figure 4b gives a big step operational semantics for the language defined in Figure 4a. The judgement $a \vdash e \Downarrow v$ indicates that expression e evaluates to value v when its argument `Arg` is bound to the stateful value a. We define the relaxed equivalence relation $\cong$ with respect to this semantics, i.e., $t_1 \cong t_2$ if and only if $\forall a, v.\ a \vdash t_1 \Downarrow v \iff a \vdash t_2 \Downarrow v$. The list of equivalences defined in this paper is summarized in Table 3.

The semantics is parametrized over the choice of base values and states. For example, the evaluation section of this paper defines values to be primitives like pointers and booleans, while a state is defined as a pair of heap (mapping from location to values) and a log of effects (e.g., from printing).

Notice that the evaluation rule for effectful operators (`F`) is identical to that of pure operators (`f`). It does not forbid effectful operations from duplicating or dropping state values. However, only effect-safe programs can be used for code generation.

[4]We assume a dataflow program takes a unique tuple-valued variable `Arg`.

Consider the original program in Figure 3 which performs four `Update` operations. It is equivalent with respect to our semantics to the effect-unsafe program shown in the third pane. We write both below in the syntax of Figure 4a, with a, b, c, x, y, and σ for the components `Arg[0]` through `Arg[5]`.

Effect-safe:

```
Update(c,
  Update(a, x, Update(a, x, σ)[1])[0],
  Update(b, y,
    Update(a, x, Update(a, x, σ)[1])[1])[1])
```

Effect-unsafe:

```
Update(c,
  Update(a, x, Update(a, x, σ)[1])[0],
  Update(b, y, Update(a, x, σ)[1])[1])
```

The effect-safe original program performs four updates, each one consuming the state value produced by the previous update. The effect-unsafe program instead re-uses the state value `Update(a, x, σ)[1]` in two distinct `Update` terms: the outer `Update(a, x, . . . )` that encloses it, and the `Update(b, y, . . . )` beside it. The first of those writes `x` to `a` while the second writes `y` to `b`. These programs are equivalent with respect to the semantics in Figure 4b, but the effect-unsafe program cannot be translated back to an executable program.

$\cong$ relates all pairs of program terms equal under this semantics. In practice, any e-graph $\mathcal{G}$ represents a subset of $\cong$, as long as each equality used to build $\mathcal{G}$ is sound under $\cong$. We denote this under-approximated equivalence relation represented by $\mathcal{G}$ as $\cong_{\mathcal{G}}$, which satisfies $(\cong_{\mathcal{G}}) \subseteq (\cong)$.

Authoring rules. An optimization rule over the e-graph is sound exactly when the terms it equates agree under the semantics of Figure 4b. This is the only requirement we place on rules. A sound rule may equate an effect-safe term with an effect-unsafe one. Rules and extraction therefore work over different relations (Table 3): rules build $\cong_{\mathcal{G}}$, while extraction searches its effect-safe restriction $\equiv_{\mathcal{G}}$. That separation makes equality saturation over effectful programs possible, since an optimization such as the redundant-`Update` rule of Figure 3 adds effect-unsafe terms to the e-graph. Note that effect-safety does not rely on design decisions specific to EGGCC's IR, such as its use of regions to model control flow (Section 7.1).

4.2 Effect Safety and $\equiv_{\mathcal{G}}$

A term is effectful if it evaluates to a stateful value (and pure otherwise). This happens if the term is `Arg`, an effectful operator, or an indexing expression into the last element of a stateful tuple value. `Arg` is the only effectful term that has no child terms. Purity is a property of the value a term denotes, not of how that value is computed. In the example above, $\texttt{Update}(a, x, \sigma)[0]$ is pure — it denotes the integer previously stored at a — even though obtaining it performs an effect. An e-class is also either effectful or not, containing only effectful terms or only pure terms. An e-class that contains both is unsound, since $\cong$ does not relate any effectful term to a pure term. Nothing prevents a rule from merging an effectful term with a pure one, but such a rule is unsound by the criterion of Section 4.1, and STATEWALK DP's guarantees, like those of any extractor, hold only for e-graphs built from sound rules.

We define effect-safety and statewalks, which are effect-safe by construction, as follows:

Definition 4.1 (Statewalk)**.** For $k \geq 0$, $\boldsymbol{\omega} = \langle e_0, e_1, \ldots, e_k \rangle$ is a statewalk if:

- $e_0 = \texttt{Arg}$.
- For all $e_i \in \boldsymbol{\omega}$, e_i is effectful.
- For each pair of terms $e_i, e_{i+1} \in \boldsymbol{\omega}$, $e_i \in \mathsf{Children}(e_{i+1})$.
- For all $e_i \in \boldsymbol{\omega}$ and for all $t \in \mathsf{Children}(e_i)$, all effectful subterms of t are in $\langle e_0, e_1, \ldots, e_{i-1} \rangle$.

Definition 4.2 (Effect-safety)**.** A term t is effect-safe under statewalk $\boldsymbol{\omega}$, denoted $\mathsf{EffSafe}(t, \boldsymbol{\omega})$, if all effectful subterms of t are in $\boldsymbol{\omega}$. We also say a term t is effect-safe, denoted $\mathsf{EffSafe}(t)$, if it is effect-safe under some statewalk (i.e., $\exists \boldsymbol{\omega}.\mathsf{EffSafe}(t, \boldsymbol{\omega})$).

Note that a statewalk is uniquely determined by its last term, since the entire statewalk can be constructed by following the child relation on effectful terms. For example, consider again the initial program in Figure 3 which performs four `Update` operations. It is effect-safe and so it has a statewalk, obtained by following the child relation through every term that carries the state value. Below we name the three `Update` subterms $u1$, $u2$, and $u3$. Some subterms occur more than once; a practical e-graph implementation stores each term only once using a hash-cons data structure.

Term:

```
(a, b, c, x, y, σ) = Arg
u1 = Update(a, x, σ)
u2 = Update(a, x, u1[1])
u3 = Update(b, y, u2[1])
Update(c, u2[0], u3[1])
```

Statewalk:

```
⟨Arg, σ,
 u1, u1[1],
 u2, u2[1],
 u3, u3[1],
 Update(c, u2[0], u3[1])⟩
```

This forms a valid statewalk because $e_0 = \texttt{Arg}$, all e_i are effectful, each e_i is a child of e_{i+1}, and all effectful terms occur in the sequence. The statewalk contains more than just the four `Update` terms because every term that carries the state value is effectful and therefore belongs to it, including the indexing expressions that extract the state: σ (that is, $\texttt{Arg}[5]$) and each $ui[1]$. Pure terms such as a and $u2[0]$ are not in the statewalk; they are instead the terms the statewalk makes available for extraction (Section 6.1).

Definition 4.3. The refined equivalence $\equiv_{\mathcal{G}}$ for effect-safe programs for e-graph $\mathcal{G}$ is defined as:

$$t_1 \equiv_{\mathcal{G}} t_2 \iff t_1 \cong_{\mathcal{G}} t_2 \wedge \mathsf{EffSafe}(t_1) \wedge \mathsf{EffSafe}(t_2)$$

By transitivity of relation refinement, $(\equiv_{\mathcal{G}}) \subseteq (\cong_{\mathcal{G}}) \subseteq (\cong)$. Given an e-graph $\mathcal{G}$ and an e-class C of $\mathcal{G}$, the problem of *effect-safe extraction* is to efficiently find an effect-safe term t represented by C. Formally, we want to find a term $t \in \mathbb{T}_{\mathcal{G}}(C)$ that satisfies $\mathsf{EffSafe}(t)$. In Section 5, we show that effect-safe extraction is NP-complete. STATEWALK DP solves this problem efficiently by leveraging statewalk width (Section 6). STATEWALK DP also greedily optimizes for the cost of the extraction, so it is a practical heuristic for the *optimal effect-safe extraction* problem, which requires t to have minimal cost according to a given cost model ($\arg\min_{t:t\in\mathbb{T}_{\mathcal{G}}(C)\wedge\mathsf{EffSafe}(t)} cost(t)$).

5 NP Completeness of Effect-Safe Extraction

In this section, we show that effect-safe extraction is NP-complete. First, we show how to encode CNF-SAT as an instance of effect-safe extraction, showing that effect-safe extraction is NP-hard. Then, we show how to encode effect-safe extraction in SAT, showing that effect-safe extraction is NP-easy. The full proofs are in Section C.1 and Section C.2.

NP-hardness. The high-level approach is to encode variables in the formula as effectful terms and clauses in the formula as pure e-classes. The e-class for a clause contains one term for each literal in the clause. The effectful terms ensure that the statewalks of the root e-class correspond exactly to all possible assignments to the variables in ϕ: each statewalk includes exactly one of x or $\neg x$ for each variable x in the formula. Intuitively, variables correspond to choices of which statewalk to follow; clauses correspond to constraints on which pure terms have compatible effect orders; satisfying assignments correspond to effect-safe extractions.

Figure 5 shows an example encoding of the formula $\phi = \neg x_0 \wedge (\neg x_1 \vee x_0) \wedge (x_2 \vee x_1 \vee \neg x_2)$. An effect-safe extraction of C_R is shown in the last panel of Figure 5. Its effect-safety is witnessed by the statewalk with final term $R(g_1(\ldots), g_2(\ldots), g_3(\ldots), s(x_2(s(\neg x_1(s(\neg x_0(\texttt{Arg})))))))$. This statewalk

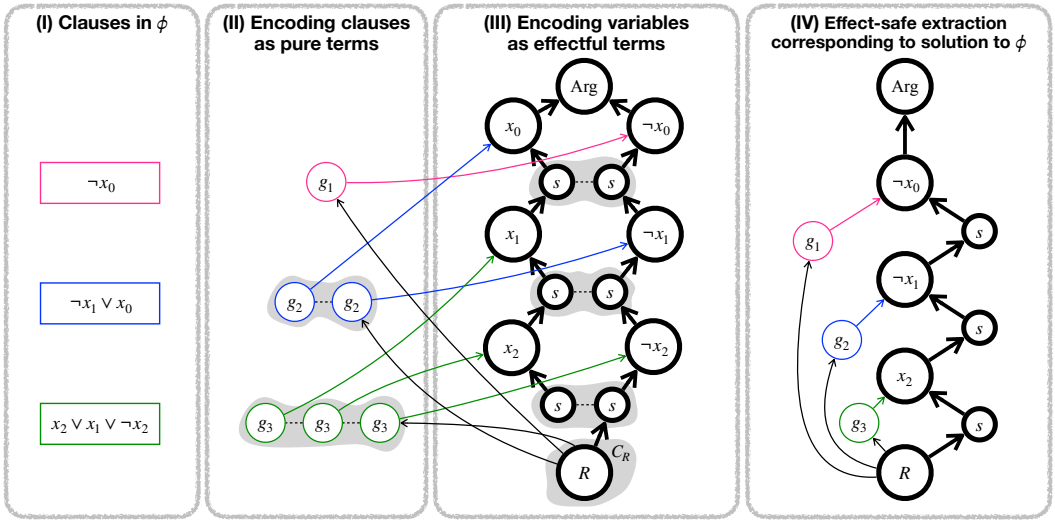


Fig. 5. E-graph encoding of the formula $\phi = \neg x_0 \wedge (\neg x_1 \vee x_0) \wedge (x_2 \vee x_1 \vee \neg x_2)$. Equivalences are shown with dotted lines, and equivalence classes are shown with a gray background. C_R denotes the e-class containing the root term $R(\ldots)$. Panel I shows the clauses in ϕ. Panel II shows the encoding of each clause as an e-class consisting of one pure term per literal in the clause. Panel III shows how the variables in ϕ are encoded using effectful terms, with auxiliary effectful terms $s(\ldots)$ ensuring independence between variable choices. Panel IV shows an effect-safe extraction of R, which corresponds to a solution to ϕ.

corresponds to the satisfying assignment of ϕ given by $\{x_0 = \bot, x_1 = \bot, x_2 = \top\}$. The extracted terms $g_i(\ldots)$ correspond to partial assignments of variables that ensure satisfaction of clause i in ϕ.

More generally, let $\phi = \varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_k$ be an arbitrary boolean formula in conjunctive normal form where $\varphi_1, \varphi_2, \ldots, \varphi_k$ are clauses over variables $x_0, x_1, \ldots, x_n$. We encode the formula using the following operators:

(1) Arg: the effectful leaf term.
(2) s: an effectful operator that takes a state value and produces a state value.
(3) x_i and $\neg x_i$: effectful operators that take a state value and produce a tuple containing a pure value and a state value.
(4) g_i: a pure operator that takes a pure value and produces a pure value.
(5) R: an effectful operator that takes k pure values and a state value.

We construct an effectful e-graph $\mathcal{G}$ as follows[5]:

(1) Add effectful terms $s(x_0(\text{Arg}))$ and $s(\neg x_0(\text{Arg}))$, denoted as t_{x_0} and $t_{\neg x_0}$.
(2) For each variable $i = 1 \ldots n$, add effectful terms $s(x_i(t_{x_{i-1}}))$ and $s(\neg x_i(s(t_{\neg x_{i-1}})))$, denoted as t_{x_i} and $t_{\neg x_i}$.
(3) Add equivalences $t_{x_i} \cong_{\mathcal{G}} t_{\neg x_i}$ for each x_i.
(4) For each clause $\varphi_i = \ell_1 \vee \ldots \vee \ell_m$, and for each literal $\ell_j \in \varphi_i$, where ℓ_j is either x_k or $\neg x_k$ for some k, add the pure term $g_i(x_k(t_{x_{k-1}}))$ or $g_i(\neg x_k(t_{\neg x_{k-1}}))$, respectively, denoted as $t_{\varphi_{i,j}}$.
(5) For each clause $\varphi_i = \ell_1 \vee \ldots \vee \ell_m$, add equivalences between all $t_{\varphi_{i,j}}$ for $j = 1 \ldots m$.
(6) Add an effectful term $R(t_{\varphi_{1,1}}, \ldots t_{\varphi_{k,1}}, t_{x_n})$; denote the e-class of this term as C_R.

[5]The dataflow IR syntax in Figure 4a would require indexing ($t[0]$ and $t[1]$) for unpacking the pure and stateful values out of tuples. These do not affect the encoding, and they are omitted for simplicity.

NP-easiness. Finding any e-graph extraction, without considering effect-safety, can be encoded as an instance of SAT using $O(n^3)$ variables and clauses, where n is the size of the e-graph. To find an effect-safe extraction, instead of finding just a single term, we solve for a statewalk, which is a sequence of terms. This can be done by cloning the single-term extraction encoding for each potential term in the statewalk and adding constraints to ensure the sequence of terms forms a statewalk. The key insight is that while there is no upper bound on the length of a statewalk, only statewalks up to length n^2 need to be considered in the search for an effect-safe extraction. As a result, it is possible to encode the effect-safe extraction problem as an $O(n^5)$ SAT instance. The formal proof, including the proof extension for optimal effect-safe extraction is in Section C.2.

Hardness does not leave an optimizer stuck, since the input program is itself an effect-safe solution, and prior work falls back to it when extraction fails [38]. Our problem definition does not assume such a solution is in hand, because the goal is to find a *better* program; whether the input can instead *guide* extraction remains open.

6 Statewalk DP

For pure e-graphs, extractors can pick any term from the root e-class to give a valid solution. Finding the optimal solution with respect to a cost model is known to be NP-complete [34, 48], and existing solutions do not scale especially in the presence of cycles [12]. Therefore, the most ubiquitous and successful extraction algorithms greedily search for good terms without guaranteeing optimality. In this section we present STATEWALK DP, an algorithm with guarantees of effect-safe extraction while optimizing for cost. For a worked example, see Appendix E.

Typical extraction algorithms for pure e-graphs build extractions bottom-up, greedily constructing extractions for each e-class. This approach is complete (always finding an extraction when one exists) and terminating. Crucially, extraction algorithms for pure e-graphs leverage the *congruence* property of e-graphs to choose any term from each e-class when constructing new terms. The extractor can freely compose a function symbol and a valid term from each direct child e-class, allowing it to keep only one extracted term per e-class.

With effects, the e-graph can contain effect-unsafe terms, and the congruence property no longer holds over effect-safe terms. Therefore, by default, terms in the same e-class cannot necessarily be used interchangeably when constructing new terms. In other words, composing effect-safe terms does not necessarily lead to another effect-safe term. As a result, applying a standard extraction algorithm to effectful e-graphs does not guarantee completeness: it can fail to find an effect-safe extraction when one exists. In fact, finding *any* effect-safe extraction is NP-complete (Section 5).

We solve this problem by grouping effect-safe terms into equivalence classes, recovering the ability to choose any term within the same equivalence class.[6] We call the ability to substitute one effect-safe term for another *substitutability*. The STATEWALK DP algorithm solves the effect-safe extraction problem using a dynamic programming approach on the equivalence classes of $\equiv_{\mathrm{SW}}$, a refinement of $\equiv_{\mathcal{G}}$. Our description focuses on the completeness of STATEWALK DP. We present proof sketches for soundness and completeness in Section 6.3 and a complexity analysis in Section 6.4.

Our STATEWALK DP implementation uses a linear cost model and produces a best-effort low cost term as in a typical effect-unaware greedy extractor. Our cost model and the optimality of STATEWALK DP's solution are discussed in Section 7.3.

6.1 Achieving Substitutability Using $\equiv_{\mathrm{SW}}$

As a first crack at STATEWALK DP's design, consider that extractions can be composed when they are effect-safe under the *same* statewalk. Intuitively, this is because the subterms all rely on the

[6] We write *equivalence class* for a class of one of the relations in Table 3, and *e-class* for the terms an e-graph stores.

same ordering of effectful operations. Every effect-safe term corresponds to a statewalk, so an extractor could exhaustively enumerate through all effect-safe terms bottom-up, storing one term per statewalk for each e-class. However, storing an extraction for every statewalk in an e-class is impractical because the number of statewalks is unbounded.

We would like to reduce the number of statewalks considered for each e-class. For pure e-graphs, the congruence property allows us to store only one term per e-class. We would like to recover a similar property for effect-safe terms, which we call substitutability. Our insight is an equivalence relation $\equiv_{\mathsf{SW}}$ which refines $\equiv_{\mathcal{G}}$. This equivalence relation also allows us to recover substitutability, the ability to substitute one effect-safe term for another when constructing new terms. With substitutability we will store dramatically fewer statewalks.

First we define the extractable set of a statewalk, which we'll use to group statewalks into equivalence classes under $\equiv_{\mathsf{SW}}$. Informally, the extractable set of statewalk $\boldsymbol{\omega}$ is the set of pure e-classes in $\mathcal{G}$ which contain an effect-safe term under $\boldsymbol{\omega}$. More precisely:

Definition 6.1 (Extractable Set)**.**

$$\mathsf{es}(\boldsymbol{\omega}) = \{\mathbb{C}_{\mathcal{G}}(t) \mid \mathsf{pure}(t) \wedge \mathsf{EffSafe}(t, \boldsymbol{\omega})\}$$

Note that any effect-safe effectful term exactly corresponds to the statewalk constructed from its effectful subterms, so for a term t, $\mathsf{es}(t)$ denotes the extractable set of the statewalk defined by t.

$\equiv_{\mathsf{SW}}$ equates any two effect-safe effectful terms, t_1 and t_2, that are equivalent under $\equiv_{\mathcal{G}}$ and induce statewalks with the same extractable set.

Definition 6.2 (Statewalk Equivalence, $\equiv_{\mathsf{SW}}$)**.**

$$t_1 \equiv_{\mathsf{SW}} t_2 \iff \mathsf{effectful}(t_1) \wedge \mathsf{effectful}(t_2) \wedge t_1 \equiv_{\mathcal{G}} t_2 \wedge \mathsf{es}(t_1) = \mathsf{es}(t_2)$$

By this definition, $\equiv_{\mathsf{SW}} \subseteq \equiv_{\mathcal{G}}$, so solving the extraction problem for $\equiv_{\mathsf{SW}}$ solves the effect-safe extraction problem of $\equiv_{\mathcal{G}}$. Because equivalence classes of $\equiv_{\mathcal{G}}$ have a one-to-one correspondence with e-classes of $\mathcal{G}$ that are inhabited by effect-safe terms, each pair $(\mathbb{C}_{\mathcal{G}}(t), \mathsf{es}(t))$, corresponds to one partition produced by $\equiv_{\mathsf{SW}}$'s refinement of $\equiv_{\mathcal{G}}$. We use these pairs as DP states in STATEWALK DP.

Under $\equiv_{\mathsf{SW}}$, effectful terms are equivalent exactly when the statewalks they define are substitutable for constructing new effect-safe terms.

Lemma 6.1 (Substitutability Within Effectful Term). *Given*

- *effect-safe effectful terms* $t_1 \equiv_{\mathsf{SW}} t_2$,
- *their corresponding statewalks* $\boldsymbol{\omega}_1 = \langle \mathtt{Arg}, \ldots, t_1 \rangle$ *and* $\boldsymbol{\omega}_2 = \langle \mathtt{Arg}, \ldots, t_2 \rangle$,
- *an extended statewalk* $\langle \mathtt{Arg}, \ldots, t_1, s \rangle$,

Then exists $s' \equiv_{\mathcal{G}} s$ *with statewalk* $\langle \mathtt{Arg}, \ldots, t_2, s' \rangle$,

We construct s' by substitution. Since s extends the statewalk, t_1 is a child of s, and by Lemma A.3 it is the only effectful one, so we substitute it with $t_2 \equiv_{\mathcal{G}} t_1$. For each pure child c_i of s, we substitute it with $c_i' \equiv_{\mathcal{G}} c_i$ such that $\mathsf{EffSafe}(c_i', \boldsymbol{\omega}_2)$, which is guaranteed to exist because $\boldsymbol{\omega}_1$ and $\boldsymbol{\omega}_2$ have the same extractable set. All children of s' are effect-safe under $\boldsymbol{\omega}_2$ and s' refers only to t_2 so s' is effect-safe. Because $\mathcal{G}$ is closed under congruence, s and s' are in the same e-class of $\mathcal{G}$, so $s' \equiv_{\mathcal{G}} s$.

Lemma 6.2 (Substitutability Within Pure Term). *Given*

- *effect-safe effectful terms* $t_1 \equiv_{\mathsf{SW}} t_2$,
- *their corresponding statewalks* $\boldsymbol{\omega}_1 = \langle \mathtt{Arg}, \ldots, t_1 \rangle$ *and* $\boldsymbol{\omega}_2 = \langle \mathtt{Arg}, \ldots, t_2 \rangle$,
- *an* s *such that* s *is pure and* $\mathsf{EffSafe}(s, \boldsymbol{\omega}_1)$,

Then there exists $s' \equiv_{\mathcal{G}} s$ *such that* $\mathsf{EffSafe}(s', \boldsymbol{\omega}_2)$.

Algorithm 1 STATEWALK DP, part 1: EXTRACTMAP finds the pure terms extractable with a given statewalk

```
1:  function ExtractMap(ω)
2:      Input: a valid statewalk, ω
3:      Output: a mapping from each pure e-class to a term of the e-class that is effect-safe with ω.
4:      E ← {}
5:      for ℓ ∈ {t | t ∈ G ∧ pure(t) ∧ leaf(t)} do
6:          if E[C_G(ℓ)] = ⊥ then E[C_G(ℓ)] ← ℓ
7:      fixed_point ← false
8:      while not fixed_point do
9:          fixed_point ← true
10:         for t = f(t_1, ..., t_n) ∈ {t | t ∈ G ∧ pure(t)} do
11:             if E[C_G(t)] ≠ ⊥ then continue        ▷ Already have an effect-safe representative.
12:             if ∀ i. E[C_G(t_i)] ≠ ⊥ ∨ (∃ e'_i. C_G(e'_i) = C_G(t_i) ∧ e'_i ∈ ω) then
13:                                  ▷ Each pure t_i can be substituted with a term that is effect-safe under ω;
14:                                                ▷ the effectful t_i (if present) can be substituted by e'_i ∈ ω.
15:                 for t_i ∈ {t_1, ..., t_n} do
16:                     t'_i ← if pure(t_i) then E[C_G(t_i)] else e'_i        ▷ At most one t_i is effectful.
17:                 E[C_G(t)] ← f(t'_1, ..., t'_n)
18:                 fixed_point ← false
19:     return E
```

This follows directly from the definition of extractable set. Because s is pure and $\mathrm{EffSafe}(s, \boldsymbol{\omega}_1)$, its e-class $\mathbb{C}_{\mathcal{G}}(s)$ belongs to $\mathrm{es}(\boldsymbol{\omega}_1)$. Since $t_1 \equiv_{\mathrm{SW}} t_2$ gives $\mathrm{es}(\boldsymbol{\omega}_1) = \mathrm{es}(\boldsymbol{\omega}_2)$, that e-class also belongs to $\mathrm{es}(\boldsymbol{\omega}_2)$, which by definition means it contains a term s' with $\mathrm{EffSafe}(s', \boldsymbol{\omega}_2)$. Both terms are effect-safe and share an e-class, so $s' \equiv_{\mathcal{G}} s$. Note that the effectful child of a pure s need not be t_1: it may be any earlier element of $\boldsymbol{\omega}_1$, as with $u2[0]$ in the example of Section 4.2.

With substitutability recovered, it suffices for STATEWALK DP to store a single statewalk per equivalence class of $\equiv_{\mathrm{SW}}$. For a specific e-class, the number of statewalk equivalence classes is at most 2^n because each equivalence class must correspond to a different extractable set and there are only 2^n possible extractable sets. Thus, the number of equivalence classes of $\equiv_{\mathrm{SW}}$ is at most $n \cdot 2^n$, and STATEWALK DP is guaranteed to terminate.

6.2 STATEWALK DP Algorithm

Algorithm 1 and Algorithm 2 show the two components of STATEWALK DP. Algorithm 2 is the entry point to the algorithm which takes an e-graph and an effectful root e-class and returns an effect-safe extraction for the given e-class.

EXTRACTMAP maps a statewalk to its extractable set. It is very similar to a standard extraction algorithm that iteratively constructs extracted terms for pure e-classes via bottom-up enumeration. The key difference is that it only uses the members of the input statewalk as effectful subterms to construct new pure terms. When there are multiple subterms in the statewalk for a particular effectful e-class, any of them can be used as the representative. By construction, the representative term for each pure e-class is effect-safe with respect to the provided statewalk.

EXTRACT is the main extraction loop. It keeps a worklist of equivalence classes of $\equiv_{\mathrm{SW}}$, whose representative statewalks can potentially be extended to construct new statewalks. The DP table keeps track of one statewalk per equivalence class, indexed by an effectful e-class and an extractable set for the statewalk. For each statewalk processed by the worklist, we enumerate all possible

Algorithm 2 STATEWALK DP, part 2: EXTRACT, the entry point, returns an effect-safe term for an effectful e-class

```
1:  function Extract(c_r)
2:      Input: an effectful e-class c_r.
3:      Output: an effect-safe term represented by c_r.
4:      DP ← {}                          ▷ Store a statewalk for each effectful e-class and extractable set
5:      e ← Keys(ExtractMap(⟨Arg⟩))                ▷ pure e-classes extractable with ⟨Arg⟩
6:      DP[ℂ_G(Arg)][e] ← ⟨Arg⟩
7:      worklist ← new queue             ▷ Worklist stores (effectful e-class, extractable set) pairs
8:      worklist.insert((ℂ_G(Arg), e))
9:      while not worklist.empty() do
10:         (c, es) ← worklist.pop()
11:         ⟨e_0, ..., e_k⟩ ← DP[c][es]                                   ▷ ℂ_G(e_k) = c
12:         eff_deps ← {s | s = g(s_1, ..., s_m) ∈ G ∧ ¬pure(s) ∧ ∃ j. ℂ_G(s_j) = c}
13:         for t = f(t_1, ..., t_n) ∈ eff_deps do
14:             if ∀ i. pure(t_i) ⟹ ℂ_G(t_i) ∈ es then        ▷ Can extend ⟨e_0, ..., e_k⟩ with t.
15:                 M ← ExtractMap(⟨e_0, ..., e_k⟩)
16:                 for t_i ∈ {t_1, ..., t_n} do
17:                     t'_i ← if pure(t_i) then M[ℂ_G(t_i)] else e_k   ▷ ¬pure(t_i) ⟹ ℂ_G(t_i) = c
18:                 t' ← f(t'_1, ..., t'_n)                  ▷ t' equivalent to t and effect-safe.
19:                 ω' ← ⟨e_0, ..., e_k, t'⟩
20:                 es' ← Keys(ExtractMap(ω'))
21:                 if DP[ℂ_G(t)][es'] = ⊥ then
22:                     DP[ℂ_G(t)][es'] ← ω'
23:                     worklist.insert((ℂ_G(t), es'))
24:     if DP[c_r] = ⊥ then return ⊥
25:     Choose es such that DP[c_r][es] ≠ ⊥ and cost is minimized.
26:     ⟨e_0, ..., e_k⟩ ← DP[c_r][es]
27:     return e_k                                  ▷ the extracted term for e-class c_r
```

extensions of the statewalk (effectful e-classes that take the last term in the current statewalk as input). When a new statewalk is constructed, its extractable set is computed using ExtractMap. If the statewalk and its extractable set form a new equivalence class under $\equiv_{SW}$, the statewalk is added to the worklist. The algorithm terminates when all items in the worklist have been processed, at which point, all equivalence classes of $\equiv_{SW}$ have been identified. If there is an entry in the DP table for the root e-class, it corresponds to an effect-safe extraction. Otherwise, no effect-safe extraction exists. Though not shown in the algorithm pseudocode, the implemented algorithm keeps the lowest cost representative statewalk for each equivalence class and uses a priority queue for the worklist, as in Dijkstra's shortest path algorithm.

6.3 STATEWALK DP Algorithm Correctness

In this section, we give a proof sketch for the soundness and completeness of STATEWALK DP. The full proof is in Appendix B.

ExtractMap is sound. Every entry in the mapping is effect-safe with ω. All terms added to the mapping are effect-safe by construction because effect-safe terms with the same statewalk are composable. The algorithm leverages this composability to construct the mapping via bottom-up enumeration starting from the leaves.

ExtractMap is complete. If an effect-safe term exists for a given e-class with $\boldsymbol{\omega}$, then the mapping will contain an entry for that e-class. All effect-safe terms must be composed from effect-safe subterms with the same statewalk, so bottom-up enumeration until fixed-point ensures that an effect-safe term for every e-class in the extractable set of $\boldsymbol{\omega}$ will be found. Soundness and completeness are both proven by induction on the depth of the smallest term in each e-class.

Extract is sound. Soundness is implied by the following inductive invariant: each entry in statewalk DP's DP table is a valid statewalk for the corresponding (e-class, extractable set). Since DP entries are never changed after they are set, and since the DP table is initialized with a valid statewalk ⟨Arg⟩, it suffices to show that for each loop iteration that assigns to DP, $\boldsymbol{\omega'}$ is a valid statewalk by construction that extends a statewalk that was already present in DP. We prove the invariant by induction on the depth of the smallest term in each e-class.

Extract is complete. Completeness is implied by the following inductive invariant: the (e-class, extractable set) pair corresponding to every valid statewalk is eventually added to the worklist. The (e-class, extractable set) pairs in the worklist correspond to the equivalence classes of $\equiv_{\mathrm{SW}}$. Storing a single statewalk for each pair suffices because of substitutability under equivalent statewalks. Every valid statewalk can be built via some sequence of statewalks processed by the worklist. Extract builds statewalks by bottom-up enumeration until fixed-point, ensuring that the equivalence class of every valid statewalk is represented in the DP table. We prove the invariant by induction on the statewalk.

6.4 Complexity

As mentioned above, statewalk DP terminates because the number of equivalence classes of $\equiv_{\mathrm{SW}}$ is finite. Despite being exponential in the worst case (as in the construction in Section 5), we observe in practice that the number of equivalence classes of $\equiv_{\mathrm{SW}}$ is generally small. We capture this insight with the parameter statewalk width.

Informally, statewalk width measures the maximum number of equivalence classes in $\equiv_{\mathrm{SW}}$ corresponding to a single equivalence class of $\equiv_{\mathcal{G}}$. This is equivalent to the maximum number of different extractable sets for terms in one equivalence class of $\equiv_{\mathcal{G}}$. More formally, we have:

Definition 6.3 (Statewalk Width). $\mathrm{sww}(\equiv_{\mathcal{G}}) = \max_{\mathrm{effectful}(t)} \left|\{\mathrm{es}(t') \mid t' \equiv_{\mathcal{G}} t\}\right|$

By this definition, we obtain the exponential upper bound:

Lemma 6.3 (Upperbound on statewalk width). $\mathrm{sww}(\equiv_{\mathcal{G}}) \leq 2^n$, *where n is the number of equivalence classes of $\equiv_{\mathcal{G}}$*

Let $k = \mathrm{sww}(\equiv_{\mathcal{G}})$ and n be the number of representative terms in $\mathcal{G}$, the complexity for statewalk DP finding any effect-safe extraction is $O(kn^2)$. If we augment the algorithm to minimize cost, the complexity grows by a logarithmic factor.

The complexity of processing each representative term in a topological order exactly once is $O(n)$, resembling a standard effect-unaware extraction algorithm. Similarly, the complexity of the Algorithm 1 is $O(n)$, traversing $\mathcal{G}$ once.

In Algorithm 2, each effectful representative term of $\mathcal{G}$ is processed at most $\mathrm{sww}(\equiv_{\mathcal{G}})$ times. This is because the representative term has only one effectful direct child, whose e-class can correspond to at most $\mathrm{sww}(\equiv_{\mathcal{G}})$ many equivalence classes of $\equiv_{\mathrm{SW}}$. So the number of calls to Algorithm 1 is bounded by $O(kn)$ over all representative terms. Thus, the complexity of statewalk DP is $O(kn^2)$.

As we will see in Section 8, statewalk width is low in practice. We hypothesize that optimization rules tend to preserve extractable sets across different statewalks. For example, a rule rewriting only pure terms cannot increase statewalk width, covering many peephole optimizations. For rules

that do rewrite effectful terms, only rarely does an application drop information (i.e., extractable e-classes) between statewalks.

7 Implementation

This section discusses the implementation of STATEWALK DP, including performance optimizations and an extension to support control flow. Finally, we describe EGGCC, an e-graph-based optimizer for effectful programs with control flow, in which we instantiate STATEWALK DP.

7.1 Control Flow and Regions

The algorithm in Section 6 operates over a general dataflow IR, which does not support control flow. To handle control flow, we extend STATEWALK DP to work with the Regionalized Value State Dependence Graph (RVSDG) representation [31]. RVSDGs represent control flow using *regions*, which are subgraphs with well-defined inputs and outputs. Each region defines a scope for its contained nodes. Nodes only refer to other nodes in the same region and never refer to nodes in other regions directly. A node in an RVSDG can be simple (e.g., those in Section 4) or structural. Structural nodes contain sub-regions. Loops are structural nodes with the loop body as a sub-region, and conditionals are structural nodes with sub-regions for each branch. This gives a hierarchical representation of control flow as nested regions.

Rewriting runs over the whole e-graph, in which a region is an ordinary node: an argument of an `If` or `DoWhile`. To extract a program, however, we decompose the e-graph into *regionalized* e-graphs, each representing terms in a single region. This step is necessary because STATEWALK DP assumes all effectful terms have a single effectful child, but branching constructs have multiple effectful children (e.g., the then and else branches of an if term). Each regionalized e-graph is constructed by replacing subterms of a representative term referring to regions (such as a loop body) with symbolic placeholders. To extract, we first run STATEWALK DP over the regionalized e-graph that contains the target e-class, which returns an extraction that may contain placeholders. For each placeholder, we recursively extract from a regionalized e-graph rooted at the placeholder's e-class. The full extraction for the root e-class is reconstructed from the extractions for each sub-region by substituting the extracted term for each placeholder. Appendix G illustrates the decomposition on a small example.

Regionalized e-graphs break extraction into smaller subproblems, but a single regionalized e-graph often still contains tens of thousands of terms, and Section 8 shows this is not enough to make ILP-based extraction tractable for EGGCC's e-graphs.

7.2 STATEWALK DP Optimizations

To improve performance, our implementation of STATEWALK DP includes several optimizations. We briefly explain the most impactful ones.

Pruning regionalized e-graphs. We remove terms that are not reachable from the root e-class and uninhabited e-classes from the regionalized e-graph to reduce its size.

Implicit terms. We do not explicitly construct and save the terms in the main loop of STATEWALK DP. Instead, we mark the existence of a term and maintain hints for later reconstruction. We construct the terms explicitly only after confirming that an effect-safe term from the root e-class exists and that no lower-cost variant was discovered.

Memoization. We cache calls to ExtractMap to avoid recomputing the same query. We improve cache hit rate by leveraging the fact that query results represented by implicit terms are determined by the set of e-classes in the statewalk.

Reusing partial results with persistent B-tree. ExtractMap is called on statewalks that grow by a single term in each subsequent call, so results for the prefix of each statewalk can be

reused from previous calls. Instead of copying the entire state, we use a persistent B-tree to maintain these intermediate results. As a result, each call to ExtractMap only needs to perform updates that are introduced by the last term in the input statewalk.

Incremental hashing. We use hashes to compare two extractable sets quickly. To avoid doing a full scan of the extractable set, which would defeat the purpose of using a persistent B-tree, we use an incremental hash function inspired by bloom filters and hyper-dimensional computation.

Liveness. For each statewalk, there is a fixed set of e-classes containing effectful terms that could potentially be used to extend the statewalk. If there are no terms in this set of e-classes that depend on a pure e-class in the statewalk's extractable set, then the pure e-class can be dropped from the extractable set, reducing the number of equivalence classes explored by statewalk DP.

Associativity-commutativity (AC) mitigation. We observe occasional exponential blow up of statewalk width in our evaluation, which arises from an interaction between an optimization that collapses the ordering of read operations and Bril's implementation of multi-dimensional arrays, which leads to many densely nested load operations. Because load operations do not change the state value, every partial permutation of loads can lead to a different equivalence class in $\equiv_{SW}$. This resembles the classic Associativity-Commutativity (AC) problem of e-graphs [2, 28]. In our implementation, we use sound heuristics to identify e-classes that might cause this blow up and use a fixed order of exploring extensions to statewalks from these e-classes. This performance optimization drastically reduces the number of equivalence classes statewalk DP explores.

7.3 Cost Model

Returning the original program is a trivial but undesirable solution to the extraction problem. To find a term that corresponds to an optimized version of the input program, we use a linear cost model that assigns a fixed constant cost to each function symbol. The cost of a term is the sum of the costs of all function symbols it contains. This cost model is used to pick the extraction used for each e-class. The cost for each sub-region in a regionalized e-graph is estimated using a traditional greedy extractor that does not guarantee effect-safety.

Like traditional greedy extractors, statewalk DP is not guaranteed to find the globally optimal term under the cost model because it greedily makes locally optimal decisions. However, it does not lose more optimality than a standard greedy extractor. That is, if a greedy extractor that does not consider effects happens to find an effect-safe term, statewalk DP's solution will be at least as good: statewalk DP considers strictly more potential extractions per e-class than a traditional greedy extractor (since it tracks one extraction per equivalence class under $\equiv_{SW}$, which corresponds to a set of extractions per e-class). In practice, statewalk DP produces good extractions (Section 8).

7.4 The eggcc Program Optimizer

We implement statewalk DP in eggcc, a prototype e-graph-based optimizer for effectful programs. eggcc takes a program written in a simple LLVM-like IR called Bril [33], converts it to an RVSDG [1], and optimizes the RVSDG in Egglog [49]. We have implemented over a dozen advanced optimizations using declarative rewrite rules in Egglog that introduce complex interactions between effectful operations, making eggcc a challenging setting for effect-safe extraction. For example, eggcc performs redundant load elimination, loop invariant code motion, redundant branch elimination, dead loop elimination, and conditional invariant code motion.

After applying optimizations in Egglog, eggcc extracts an effect-safe RVSDG from the e-graph, and converts the optimized RVSDG back to Bril. During the conversion back to Bril, eggcc applies simple cleanup optimizations like eliminating unreachable branches [1]. In Section 8, we show eggcc produces programs with similar quality to LLVM.

8 Evaluation

Our evaluation answers three research questions.

RQ1 How does the performance of STATEWALK DP compare to ILP-based extraction? (Section 8.1)
RQ2 How does the runtime of extracted programs produced by STATEWALK DP compare to those produced by ILP? (Section 8.2)
RQ3 Can STATEWALK DP enable domain-specific effectful optimizations? (Section 8.3)

We show that STATEWALK DP is 520 times faster than Gurobi.[7] We then show STATEWALK DP produces programs of similar quality to LLVM and also to ILP-based extraction. Finally, we present a case study showing the benefit of domain-specific optimizations enabled by STATEWALK DP. *The key takeaway is that with STATEWALK DP, extraction is no longer the bottleneck in EGGCC's end-to-end optimization pipeline.*

Setup. We directly compare the performance of STATEWALK DP and solver-based effect-safe extraction algorithms by implementing multiple extraction techniques in EGGCC. For ILP extraction, we compare to the state-of-the-art commercial solver Gurobi and the open-source solver CBC. The ILP encoding of the extraction problem uses the same cost model as STATEWALK DP.

We run EGGCC on the Bril benchmark suite and the PolyBench benchmark suite [29] translated to Bril. The Bril benchmark suite contains 64 programs spanning geometry, matrix operations, numerical methods, graph algorithms, and bitwise operations. In total, we have 96 benchmarks.

For each benchmark, we compare STATEWALK DP against an ILP-based extraction algorithm that guarantees effect-safe extractions. All experiments were run on a machine with an AMD EPYC 7702P CPU and 512 GB of DDR4 memory. The many cores and large memory let us run multiple benchmarks in parallel, making the comparison to ILP-based extraction feasible. Individual runs of EGGCC and runs of STATEWALK DP never exceed the limit of 16 GiB of memory. We test the performance of EGGCC's binaries sequentially on a single core.

ILP encoding. For comparison to ILP, EGGCC encodes the extraction problem as an ILP instance and calls out to a (configurable) external ILP solver to find a low-cost solution, similar to prior work in Peggy [38]. Each term t in the e-graph is associated with a binary variable p_t, indicating whether t is selected in the extraction. For every candidate term u for the subchild at position i of term t, a binary variable $s_{t,i,u}$ indicates whether u is selected as the i-th child of t. Constraints are added to ensure the variables form a single term represented by the root e-class and that the extraction is effect-safe. For full details of the encoding, see Appendix D.

8.1 STATEWALK DP Compared to ILP-based Extraction (RQ1)

We show that STATEWALK DP solves the bottleneck of effect-safe extraction. We investigate the effect of factors including individual regionalized e-graphs, a large program, ILP encoding size, and statewalk width. In every case, STATEWALK DP outperforms ILP-based extraction, which is infeasible in practice.

Across all benchmarks, EGGCC spends an average of 7.925 seconds outside of extraction, with an average of 1.536 seconds spent in its extraction phase[8]. STATEWALK DP never times out. For comparison, LLVM at `-O3` takes an average of 0.834 seconds to compile the same programs. That EGGCC is

[7] Speedup is computed as the geometric mean of the ratios of runtimes on each regionalized e-graph, with timeouts counted as 5 minutes. For the few infeasible points, time taken to report infeasibility is used.
[8] EGGCC invokes STATEWALK DP as a separate process, so the measured extraction phase also includes transferring the serialized e-graph across the process boundary, parsing it, and constructing the regionalized e-graphs. STATEWALK DP's own search accounts for under 3% of this phase on average.

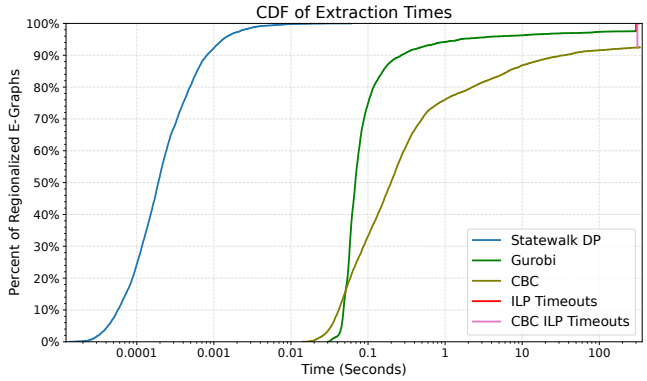


Fig. 7. Cumulative distribution function of extraction times for STATEWALK DP and ILP-based extraction, over all regionalized e-graphs across all benchmarks (including the raytracer). The x-axis is on a **log scale**. The vertical lines at the right are timeouts and infeasible extractions, which occur only for the ILP baselines: STATEWALK DP is complete (Section 6.3) and returns an effect-safe extraction for every regionalized e-graph here. Appendix D explains why the ILP encoding can report infeasibility.

slower is expected: it explores a space of equivalent programs rather than applying passes in a fixed order, which costs compile time but is what makes optimizations like those of Section 8.3 expressible.

On the other hand, ILP-based extraction is intractable for many benchmarks. On the PolyBench benchmark suite, both Gurobi and CBC *time out on all 30 benchmarks after 5 minutes* for extracting a single regionalized e-graph. A longer timeout is not feasible because benchmarks in the PolyBench suite produce an average of 209 regionalized e-graphs. In the Bril benchmark suite, Gurobi times out on 17 benchmarks. On benchmarks where Gurobi does not time out, EGGCC spends an average of 0.360 seconds running STATEWALK DP compared to 26.211 seconds running ILP-based extraction. Timeouts in Gurobi's solving do not seem to correlate with the size of the e-graph or the size of the resulting ILP encoding. Figure 6 shows the relationship between e-graph size and ILP encoding size.

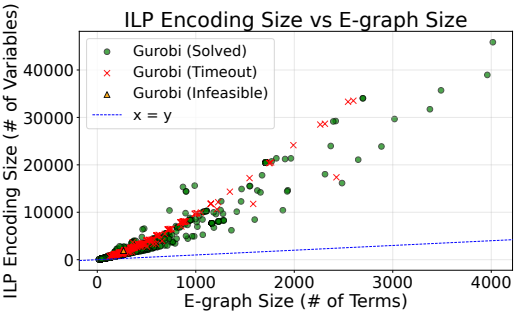


Fig. 6. Scatter plot showing ILP encoding size compared to e-graph size. Encoding size is measured in number of variables, which also bounds the number of constraints. E-graph size is measured in number of representative terms.

The number of variables in the encoding is 4.55 times the size of the e-graph on average (geometric mean), with a maximum of 17.63 times. This shows that despite a reasonable ILP encoding scaling with problem size, ILP-based extraction remains unreliable.

Region-level analysis. We perform a finer-grained comparison of STATEWALK DP and ILP-based extraction by measuring their runtimes on all 9610 regionalized e-graphs produced by EGGCC across all benchmarks. ILP-based extraction takes an average of 0.965 seconds per regionalized e-graph, and times out on 234 after 5 minutes. STATEWALK DP extraction takes an average of 0.000426 seconds

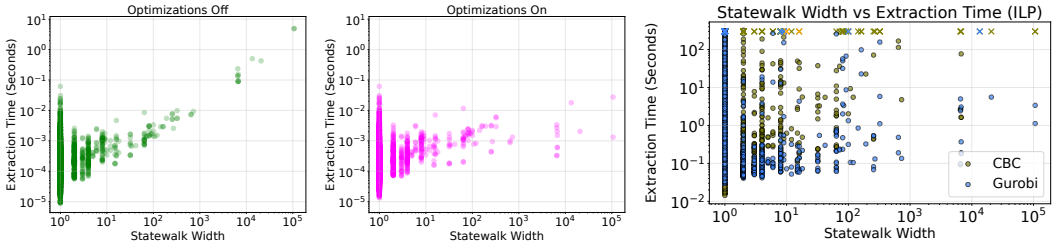


Fig. 9. Extraction time against the statewalk width of the regionalized e-graph, for STATEWALK DP without its optimizations (left), STATEWALK DP with them (middle), and ILP-based extraction (right). STATEWALK DP's optimizations are liveness analysis and AC mitigation. **All axes are logarithmic**; on linear axes almost every point would land on top of another. The two STATEWALK DP panels share axis limits, so they are directly comparable. Without optimizations, extraction time grows with statewalk width, reaching 4.9 seconds at the largest widths; with them it stays below 0.0619 seconds throughout, leaving the upper right empty. ILP-based extraction shows no such trend but times out (crosses) at every statewalk width. The dense column at the left of each panel is the 94.7% of regions with statewalk width 1.

per regionalized e-graph and spends a maximum of 0.0619 seconds on any single regionalized e-graph. Figure 7 shows the distribution of extraction times for STATEWALK DP and ILP-based extraction on all the regionalized e-graphs. A point (t, p) means a fraction p of extractions finished within time t. STATEWALK DP completes 92% in under a millisecond and all of them within 0.0619 seconds; at 0.1 seconds Gurobi has completed 75% and CBC 33%. The ILP baselines exhaust the 5-minute timeout on a tail of difficult extractions.

The experiments in this section run with the same cost model for STATEWALK DP and ILP-based extraction. The ILP-based extraction finds an optimal solution (modulo the incompleteness of the encoding). However, even when disregarding cost completely, we find that Gurobi still times out on 4 benchmarks and remains a bottleneck, spending an average of 30.589 seconds running ILP-based extraction across all benchmarks. This suggests that the bottleneck of ILP-based extraction is not optimizing cost, but rather finding any effect-safe extraction at all.

Case study on a large program. We also run EGGCC on a raytracer compiled from 821 lines of code written in a subset of Rust to 2503 lines of Bril code. EGGCC spends 171.605 seconds performing equality saturation on the raytracer and 22.603 seconds in its extraction phase. Of that phase, STATEWALK DP's own search accounts for only 0.598 seconds, summed over all 724 regionalized e-graphs that EGGCC produces while optimizing the raytracer; the remainder is the serialization and process overhead noted above. The raytracer produces large regionalized e-graphs, totaling 158,593 terms after pruning (Section 7.2), with a maximum of 4,016 terms in a single regionalized e-graph. Despite the large size, STATEWALK DP still takes a maximum of 0.018141 seconds to extract from any single regionalized e-graph. ILP-based extraction, in contrast, times out on 35 of the 724 regionalized e-graphs after 5 minutes.

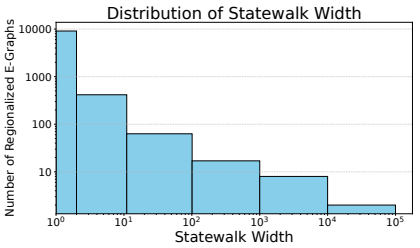


Fig. 8. The distribution of statewalk width in the Bril benchmark suite and the PolyBench benchmark suite. 94.7% of regions have a statewalk width of 1, and 98.4% have a statewalk width of 5 or less.

STATEWALK DP and ILP Extraction Time Scaling. We study how the extraction time of STATEWALK DP and ILP-based extraction changes with respect to statewalk width. Figure 8 shows the

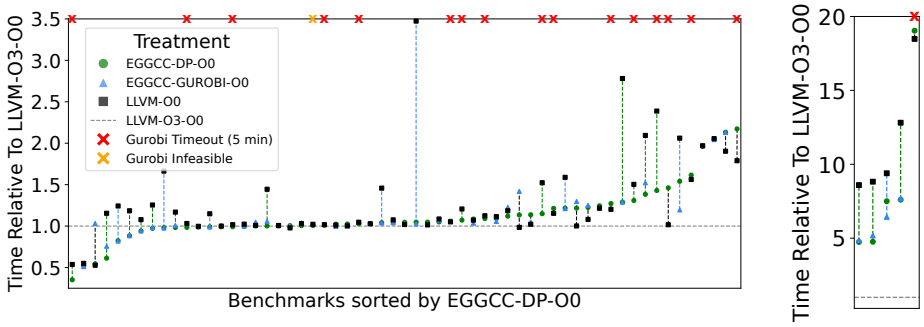


Fig. 10. Comparison of EGGCC to LLVM on the Bril benchmark suite. Lower numbers are better. The y-axis shows the ratio between the average runtime and the average runtime of the baseline (LLVM-O3-O0). The grey line at 1.0 indicates the baseline. The vertical lines indicate the difference between the worst and best treatment for each benchmark. The graph is split into two parts due to 5 outlier benchmarks for which LLVM-O3-O0 gave large speedups.

empirical measurements of statewalk width in the e-graph produced by EGGCC on the Bril benchmark suite and PolyBench benchmark suite. 94.7% of regions have a statewalk width of 1, while 98.4% have a statewalk width of 5 or less. Two outliers have a statewalk width of 106,259. These outliers come from the PolyBench "heat-3d" benchmark, which computes heat diffusion in a large grid. The statewalk width is large due to a series of memory accesses with no data dependencies between them, enabling associativity and commutativity between them. Our Associativity-Commutativity (AC) mitigation optimization reduces this factor significantly, making the maximum size of the dynamic programming table for a single e-class a manageable 24, despite the colossal statewalk width.

Figure 9 shows how the extraction time of STATEWALK DP and ILP-based extraction scales with respect to statewalk width. The extraction time of ILP-based extraction does not seem to correlate with statewalk width, while STATEWALK DP's extraction time without optimizations does. With STATEWALK DP's optimizations enabled, it scales well even in the presence of large statewalk widths.

8.2 Output Quality: Runtime of Extracted Program (RQ2)

A head-to-head comparison between STATEWALK DP and ILP-based extracted programs shows that STATEWALK DP's outputs are on par with ILP (when ILP is feasible). We also show EGGCC's e-graphs are non-trivial, rich enough to produce significant speedups over the baseline, even compared to LLVM's optimizations.

To measure the quality of STATEWALK DP's extracted programs, we compare four treatments across the Bril and PolyBench benchmarks:

- `LLVM-O0`: run `clang-o0` on the input program.
- `LLVM-O3-O0`: runs `llvm-o3` and `clang-o0` on the input program.
- `EGGCC-DP-O0`: runs `llvm-o0` and `clang-o0` on the output of EGGCC using STATEWALK DP.
- `EGGCC-GUROBI-O0`: runs `llvm-o0` and `clang-o0` on EGGCC output using ILP extraction with Gurobi.

To run LLVM, we convert Bril programs to LLVM by using the "Brillvm" tool from the developers of Bril. In the case of `LLVM-O3-O0`, we run LLVM with optimization level O3 and generate code with optimization level O0, disabling target-specific optimizations. Disabling target-specific

optimizations ensures fair comparison to EGGCC, which is target-agnostic. We measure the runtime of the resulting binaries by using x86's `rdtsc` instruction to count CPU cycles taken to execute the program. Each resulting binary is run 200 times and we report the average runtime. Appendix F gives the absolute runtimes behind the normalized numbers reported here, along with their standard deviations.

Brillvm uses the `alloca` instruction to allocate registers, which is a common way to compile imperative programs to LLVM [19]. For all treatments, we therefore enable LLVM's `sroa` pass which scalarizes memory allocations [21]. We also enable register allocation related optimizations for all treatments using the flags `-regalloc=greedy` and `-optimize-regalloc` since EGGCC does not do machine-level optimizations.

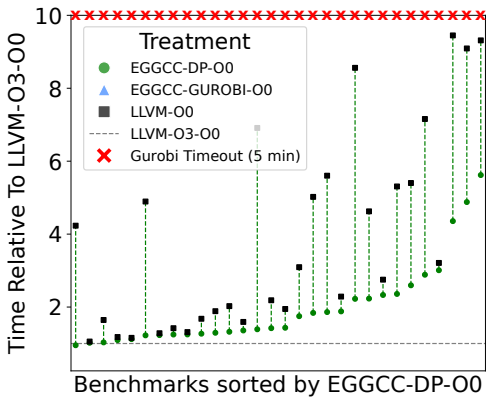


Fig. 11. Comparison of EGGCC to LLVM on the PolyBench benchmark suite. Lower numbers are better.

Figure 10 and Figure 11 compare the performance of the resulting binaries from EGGCC with different extraction algorithms to those from LLVM. We show that EGGCC with STATEWALK DP performs similarly to EGGCC with ILP-based extraction. On the PolyBench benchmark suite, normalized to `LLVM-O0`, EGGCC achieves a geometric mean time across all benchmarks of 0.571, compared to 0.326 for `LLVM-O3-O0`. On the Bril benchmark suite, EGGCC achieves a mean normalized time of 0.876 compared to 0.695 for `LLVM-O3-O0`. We also tried LLVM with target-specific optimization on, which achieves 0.312 on PolyBench and 0.542 on Bril. This matches our expectation since EGGCC is a prototype compiler and misses many optimizations present in LLVM. Spot-checking the generated assembly, the largest gaps are scalar promotion of memory, which leaves redundant loads and stores, along with generalized loop unrolling and stack coloring; none of these is a fundamental obstacle.

8.3 Case Study: Hacker's Delight (RQ3)

To showcase the impact of performing domain-specific effectful optimizations in e-graphs, made possible by STATEWALK DP, we perform a case study using an optimization taken from Hacker's Delight [41], a collection of efficient implementation tricks for low-level, bit-manipulating arithmetic.

The Hacker's Delight trick we implement is a one-liner for the `lowbit` function. For a positive integer n, `lowbit(n)` returns the least significant 1-bit of n. This function is used in bit-manipulation and data structures like Fenwick trees [9], which are used to support operations on statistical frequency tables. We show the naïve implementation and the one-liner[9] on the right.

```
lb = 1
while n % (lb * 2) == 0 {
  lb = lb * 2
}
return lb
```

```
return n & (-n)
```

Neither LLVM nor GCC performs this optimization. Adding optimizations like this to a traditional compiler is extremely delicate, requiring deep knowledge of existing passes to mitigate phase-ordering problems and careful pattern matching to perform the optimization reliably. Our rules sidestep both difficulties by

[9]Why this works: Modern machines use 2's complement for binary representation of signed integers. $-n$ in 2's complement effectively flips all the bits of n except for the least significant 1 bit and the trailing zeros. The bitwise and instruction then picks out the only bit that is 1 in both numbers, which is the lowest 1 bit of n.

running alongside EGGCC's other rules: the e-graph already holds the syntactic variants that a hand-written pattern would have to anticipate.

Implementation. We implemented the `lowbit` optimization as three additional EGGLOG rules, 36 lines excluding comments, without modifying any existing EGGCC rules. The first rule marks predicates that test whether a value is even. The second uses it to identify loops that iterate over the trailing zeros of a variable n. The third detects an output of such a loop equivalent to `lowbit(n)`, marking it equivalent to the optimized implementation. Appendix H gives all three. These rules cooperate seamlessly with the existing analyses and optimizations of EGGCC, including effectful optimizations. As such, the rules were easy to develop.

To showcase the power of this optimization, we implemented the Fenwick tree data structure in Rust, whose core operations involve calling the naïve implementation in a loop. The program performs $2n$ operations on a tree of size n, with $n = 2 \times 10^5$.

Results. The runtime performance of the Fenwick tree implementation under different optimization treatments is shown in Figure 12.

- `EGGCC-DP-O0` (leftmost, light blue) includes the Hacker's Delight rules alongside the rest of EGGCC's optimizations. Applying the Hacker's Delight optimization to the Fenwick tree eliminates the entire hot loop, yielding a 6.52 times speedup compared to `LLVM-O3-O3` and a 23.01 times speedup compared to `LLVM-O3-O0`.
- `LLVM-O3-O3` (second, yellow) runs `clang-o3`. This treatment replaces divisions and multiplications with bitwise shifts but could not further optimize the while loop.
- `LLVM-O3-O0` and `LLVM-O0` (middle, purple and black) are as described in Section 8.2.
- `EGGCC-DP-NOHACKER-O0` (rightmost, green) runs EGGCC without the Hacker's Delight rules.

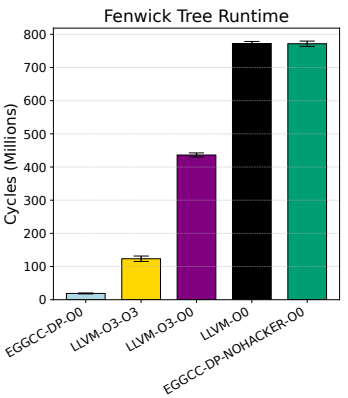


Fig. 12. Fenwick tree runtime in CPU cycles. Lower is better.

This comparison illustrates the potential of applying domain-specific optimizations to effectful programs enabled by equality saturation and STATEWALK DP.

8.4 Demonstrating the Effectful E-graph Extraction Bottleneck in Peggy

Figure 1 in Section 1 shows the results of running Peggy and EGGCC with ILP-based extraction to demonstrate that effect-safe extraction is a bottleneck for e-graph-based program optimizers. Here, we describe the experimental setup for Peggy. We ran Peggy on (1) a subset of 11 SpecJVM programs, which Peggy was originally evaluated on, and (2) 30 programs in PolyBench, which we manually translated to Java. By default, Peggy optimizes methods independently and does not perform inlining, while EGGCC does. To derive benchmarks with larger method bodies, we inlined methods in Peggy and ran Peggy on both the original programs and the inlined programs. In total, we optimized 281 methods using Peggy. We use Java bytecode instruction count as the measure of program length for Peggy. We measure the percentage of time spent in e-graph extraction, with a timeout of 10 minutes. Peggy timed out on 19 of the 281 methods, and extraction became a bottleneck quickly when the programs got larger. The results are shown in Figure 1 (Left).

9 Related Work

Effect-safe Extraction in Peggy. In this paper, we compare our approach to Peggy [38], which uses a succinct dataflow-based IR called Program Expression Graphs (PEG) for program representation. In Peggy, translating the extracted PEG back to a CFG (for code generation) can fail and is difficult to implement, as documented in a separate 37-page technical report [39]. Peggy did not study the effect-safe extraction problem theoretically. Due to the incompleteness of its ILP encoding, it cannot guarantee the optimality of its extraction, even when ILP succeeds. Despite ILP being NP-complete, the encoding does not imply that effect-safe extraction is in NP.

Pure Extraction in Practice. The pure extraction problem is commonly handled by algorithms that produce best-effort results [4, 42, 46]. While ILP solvers can theoretically produce optimal outputs, they scale poorly and restrict the cost model to those that can be expressed as systems of linear equations. The only practical optimal pure extraction algorithm is based on treewidth [12, 36], but it only works on sparse e-graphs and struggles in the presence of cycles.

A host of related work sidesteps the problem of effect-safe extraction using pure extraction. DialEgg optimizes MLIR with e-graphs [47]. DialEgg does not directly address problems with effects, but suggests that users take care in the presence of effects. Cranelift is a production-grade JIT compiler that uses an e-graph-like data structure for optimization [8]. To deal with effects, Cranelift restricts optimizations to pure terms, relying on a complex skeleton structure to preserve the order of effectful operations. Similarly, the e-graph-based Julia program optimizer uses a CFG skeleton structure to preserve effectful operations [22]. HEC [45] is a framework for equivalence checking using e-graphs that is able to handle memory and control-flow transformations, but since it performs equivalence checking, not optimization, the tool does not need to support effect-safe extraction.

10 Conclusion and Future Work

Effect-safe extraction has been a practical bottleneck for effectful e-graph optimizers for more than a decade [38]. This paper addresses that challenge with STATEWALK DP, a tractable effect-aware extraction algorithm parameterized by statewalk width. Our results show that STATEWALK DP removes extraction as a bottleneck while producing high-quality effect-safe programs. Beyond extraction, statewalk width offers a new lens on effectful interactions between terms in an e-graph, exposing structure that may be useful for analyses, verification, or effect-aware rewrite strategies.

More broadly, eliminating ILP from effectful extraction enables equality saturation in compiler and superoptimizer settings. Many existing IRs mix pure computation with effectful operations. Examples include bytecode formats as in WebAssembly [13], region-based pipelines in MLIR [23], SSA-based forms in LLVM IR [20] or Swift SIL [37], and last-stage IRs like OCaml's Cmm [26]. STATEWALK DP makes it feasible to start exploring effectful e-graph optimization in these settings.

In future work, we are excited about combining statewalk width with other parameters like treewidth to tackle the optimal effect-safe extraction problem for cost-critical settings like superoptimizers. Formalizing the observations of Section 6.4 about which rules increase statewalk width may also provide stronger theoretical bounds for real programs. A further direction is to make extraction aware of distinct categories of effect, so that independent effects may commute. A notion of separation in the style of separation logic would let the state split into pieces that evolve independently, generalizing statewalk width and STATEWALK DP to statewalks that split and merge.

Data-Availability Statement

The source code of EGGCC and STATEWALK DP, together with all benchmarks and scripts used to produce the results in Section 8, is archived on Zenodo [10] at https://doi.org/10.5281/zenodo.21479729. Development continues at https://github.com/egraphs-good/eggcc.

Acknowledgments

We thank Glenn Sun for porting the PolyBench benchmarks, Kevin Yan for infrastructure work including pretty printing, and Ryan Berger for work on our nightly benchmarking infrastructure. We thank Patrick LaFontaine for the `rs2bril` compiler and Adrian Sampson for creating and maintaining Bril. We also thank the OOPSLA reviewers, whose suggestions improved both the formal development and the evaluation.

This material is based upon work supported by the National Science Foundation under Grant No. 2232339 and 2312195, and by an National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-2140004. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Helge Bahmann, Nico Reissmann, Magnus Jahre, and Jan Christian Meyer. 2015. Perfect Reconstructability of Control Flow from Demand Dependence Graphs. *ACM Trans. Archit. Code Optim.* 11, 4, Article 66 (Jan. 2015), 25 pages. doi:10.1145/2693261

[2] Dan Benanav, Deepak Kapur, and Paliath Narendran. 1985. Complexity of matching problems. In *Rewriting Techniques and Applications*, Jean-Pierre Jouannaud (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 417–429.

[3] Ian Briggs, Yash Lad, and Pavel Panchekha. 2024. Implementation and Synthesis of Math Library Functions. *Proc. ACM Program. Lang.* 8, POPL, Article 32 (Jan. 2024), 28 pages. doi:10.1145/3632874

[4] Yaohui Cai, Kaixin Yang, Chenhui Deng, Cunxi Yu, and Zhiru Zhang. 2025. SmoothE: Differentiable E-Graph Extraction. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1020–1034. doi:10.1145/3669940.3707262

[5] Fabricio Olivetti de Franca and Gabriel Kronberger. 2023. Reducing Overparameterization of Symbolic Regression Models with Equality Saturation. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Lisbon, Portugal) *(GECCO '23)*. Association for Computing Machinery, New York, NY, USA, 1064–1072. doi:10.1145/3583131.3590346

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

[7] Rod G. Downey and Michael R. Fellows. 1995. Fixed-Parameter Tractability and Completeness I: Basic Results. *SIAM J. Comput.* 24, 4 (1995), 873–921. arXiv:https://doi.org/10.1137/S0097539792228228 doi:10.1137/S0097539792228228

[8] Chris Fallin. 2023. ægraphs: Acyclic E-graphs for Efficient Optimization in a Production Compiler. https://cfallin.org/pubs/egraphs2023_aegraphs_slides.pdf. [Accessed: 2025-02-09].

[9] Peter M. Fenwick. 1994. A new data structure for cumulative frequency tables. *Softw. Pract. Exper.* 24, 3 (March 1994), 327–336. doi:10.1002/spe.4380240306

[10] Oliver Flatt, Anjali Pal, Yihong Zhang, Ryan Tjoa, Kirsten Graham, Alex Fischman, Chandrakana Nandi, Eli Rosenthal, Zachary Tatlock, and Haobin Ni. 2026. *Efficient Extraction for Effectful E-Graphs*. Zenodo. doi:10.5281/zenodo.21479729

[11] Amir Kafshdar Goharshady, Chun Kit Lam, and Lionel Parreaux. 2024. Fast and Optimal Extraction for Sparse Equality Graphs. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 361 (Oct. 2024), 27 pages. doi:10.1145/3689801

[12] Amir Kafshdar Goharshady, Chun Kit Lam, and Lionel Parreaux. 2024. Fast and Optimal Extraction for Sparse Equality Graphs. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 361 (Oct. 2024), 27 pages. doi:10.1145/3689801

[13] W3C WebAssembly Working Group. 2023. WebAssembly Core Specification. Accessed: 2025-11-13. https://www.w3.org/TR/wasm-core-2/

[14] Jakob Hartmann, Guoliang He, and Eiko Yoneki. 2024. Optimizing Tensor Computation Graphs with Equality Saturation and Monte Carlo Tree Search. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques* (Long Beach, CA, USA) *(PACT '24)*. Association for Computing Machinery, New York, NY, USA, 40–52. doi:10.1145/3656019.3689611

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

[16] Smail Kourta, Adel Abderahmane Namani, Fatima Benbouzid-Si Tayeb, Kim Hazelwood, Chris Cummins, Hugh Leather, and Riyadh Baghdadi. 2022. Caviar: an e-graph based TRS for automatic code optimization. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction* (Seoul, South Korea) *(CC 2022)*. Association for Computing Machinery, New York, NY, USA, 54–64. doi:10.1145/3497776.3517781

[17] Dexter Kozen. 1993. *Partial Automata and Finitely Generated Congruences: An Extension of Nerode's Theorem.* Birkhäuser Boston, Boston, MA, 490–511. doi:10.1007/978-1-4612-0325-4_16

[18] Prasad A. Kulkarni, David B. Whalley, Gary S. Tyson, and Jack W. Davidson. 2006. Exhaustive optimization phase order space exploration. In *International Symposium on Code Generation and Optimization (CGO'06).* IEEE Computer Society, New York, NY, USA, 306–318. doi:10.1109/CGO.2006.15

[19] Chris Lattner and Vikram Adve. 2025. LLVM Tutorial: 7. Kaleidoscope: Extending the Language: Mutable Variables. https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl07.html. [Accessed: 2025-02-05].

[20] LLVM Project. 2025. LLVM Language Reference Manual. Accessed: 2025-11-13. https://llvm.org/docs/LangRef.html

[21] LLVM Project. 2025. sroa: Scalar Replacement of Aggregates. Accessed: 2025-11-13. https://llvm.org/docs/Passes.html#sroa-scalar-replacement-of-aggregates

[22] Jules Merckx, Tim Besard, and Bjorn De Sutter. 2025. Equality Saturation for Optimizing High-Level Julia IR. arXiv:2502.17075 [cs.PL] https://arxiv.org/abs/2502.17075

[23] MLIR Project. 2025. MLIR — Multi-Level IR Compiler Framework. Accessed: 2025-11-13. https://mlir.llvm.org/

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

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

[26] OCaml Development Team. 2023. Cmm Intermediate Representation — source (cmm.mli). Accessed: 2025-11-13, commit cce52acc7c7903e92078e9fe40745e11b944f0. https://github.com/ocaml/ocaml/blob/cce52acc7c7903e92078e9fe40745e11a1b944f0/asmcomp/cmm.mli#L168

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

[28] G. PLOTKIN. 1972. Building-in Equational Theories. *Machine Intelligence* 7 (1972), 73–90. https://cir.nii.ac.jp/crid/1571698599605621504

[29] Louis-Noël Pouchet and Tomofumi Yuki. 2016. PolyBenchC-4.2.1. https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1. [Accessed: 2025-02-05].

[30] Rolph Recto and Andrew C. Myers. 2023. A Compiler from Array Programs to Vectorized Homomorphic Encryption. arXiv:2311.06142 [cs.PL] https://arxiv.org/abs/2311.06142

[31] Nico Reissmann, Jan Christian Meyer, Helge Bahmann, and Magnus Själander. 2020. RVSDG: An Intermediate Representation for Optimizing Compilers. 28 pages. doi:10.1145/3391902

[32] Brett Saiki, Jackson Brough, Jonas Regehr, Jesús Ponce, Varun Pradeep, Aditya Akhileshwaran, Zachary Tatlock, and Pavel Panchekha. 2024. Target-Aware Implementation of Real Expressions. arXiv:2410.14025 [cs.PL] https://arxiv.org/abs/2410.14025

[33] Adrian Sampson. 2019. Bril: A Compiler Intermediate Representation for Learning. https://capra.cs.cornell.edu/bril. [Accessed: 2025-02-16].

[34] Michael Stepp. 2011. Equality Saturation: Engineering Challenges and Applications. https://rosstate.org/publications/eqsat/MikeThesis.pdf

[35] Dan Suciu, Yisu Remy Wang, and Yihong Zhang. 2025. Semantic Foundations of Equality Saturation. In *28th International Conference on Database Theory (ICDT 2025) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 328)*, Sudeepa Roy and Ahmet Kara (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 11:1–11:18. doi:10.4230/LIPIcs.ICDT.2025.11

[36] Glenn Sun, Yihong Zhang, and Haobin Ni. 2024. E-Graphs as Circuits, and Optimal Extraction via Treewidth. arXiv:2408.17042 [cs.DS] https://arxiv.org/abs/2408.17042

[37] Swift Project. 2025. SIL: Swift Intermediate Language. Accessed: 2025-11-13. https://apple-swift.readthedocs.io/en/latest/SIL.html

[38] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09).* Association for Computing Machinery, New York, NY, USA, 264–276. doi:10.1145/1480881.1480915

[39] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2010. *Translating between PEGs and CFGs.* Technical Report. University of California, San Diego. http://www.cs.cornell.edu/~ross/publications/eqsat/

[40] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. 2020. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *Proceedings of the VLDB Endowment* 13, 11 (2020), 1919–1932. doi:10.14778/3407790.3407799

[41] Henry S. Warren. 2012. *Hacker's Delight* (2nd ed.). Addison-Wesley Professional.

[42] Max Willsey, Trevor Hansen, Oliver Flatt, and contributors. 2025. Extraction Gym: A suite of benchmarks to test e-graph extraction algorithms. https://github.com/egraphs-good/extraction-gym/. Accessed: 2025-02-24.

[43] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (jan 2021), 29 pages. doi:10.1145/3434304

[44] Yichen Yang, Phitchaya Mangpo Phothilimtha, Yisu Remy Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. arXiv:2101.01332 [cs.AI]

[45] Jiaqi Yin, Zhan Song, Nicolas Bohm Agostini, Antonino Tumeo, and Cunxi Yu. 2025. HEC: Equivalence Verification Checking for Code Transformation via Equality Saturation. arXiv:2506.02290 [cs.AR] https://arxiv.org/abs/2506.02290

[46] Jiaqi Yin, Zhan Song, Chen Chen, Yaohui Cai, Zhiru Zhang, and Cunxi Yu. 2025. e-boost: Boosted E-Graph Extraction with Adaptive Heuristics and Exact Solving. arXiv:2508.13020 [cs.AI] https://arxiv.org/abs/2508.13020

[47] Abd-El-Aziz Zayed and Christophe Dubach. 2025. DialEgg: Dialect-Agnostic MLIR Optimizer using Equality Saturation with Egglog. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization* (Las Vegas, NV, USA) *(CGO '25)*. Association for Computing Machinery, New York, NY, USA, 271–283. doi:10.1145/3696443.3708957

[48] Yihong Zhang. 2023. The E-graph extraction problem is NP-complete. https://effect.systems/blog/egraph-extraction.html

[49] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better Together: Unifying Datalog and Equality Saturation. *Proc. ACM Program. Lang.* 7, PLDI, Article 125 (jun 2023), 25 pages. doi:10.1145/3591239

[50] Yifan Zhao, Hashim Sharif, Vikram Adve, and Sasa Misailovic. 2024. Felix: Optimizing Tensor Programs with Gradient Descent. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 367–381. doi:10.1145/3620666.3651348