Parallel and Customizable Equality Saturation

Jonathan Van der Cruysse
jonathan.vandercruysse@mail.mcgill.ca
McGill University
Montreal, Quebec, Canada

Abd-El-Aziz Zayed
abd-el-aziz.zayed@mail.mcgill.ca
McGill University
Montreal, Quebec, Canada

Mai Jacob Peng
mai.peng@mail.mcgill.ca
McGill University
Montreal, Quebec, Canada

Christophe Dubach
christophe.dubach@mcgill.ca
McGill University & Mila
Montreal, Quebec, Canada

Abstract

Equality saturation enables compilers to explore many semantically equivalent program variants, deferring optimization decisions to a final extraction phase. However, existing frameworks exhibit sequential execution and hard-coded saturation loops. This limits scalability and requires significant engineering effort to customize saturation behavior.

This paper addresses these limitations using three novel techniques. First, it shows how saturation can be parallelized thanks to the use of thread-safe data structures and the notion of deferred e-graph updates. Second, it provides an extensible mechanism to express custom and composable saturation strategies. Third, it generalizes e-graph metadata to support custom e-graph annotations.

The implementation, written in Scala, is evaluated on four use-cases: classical program optimization, idiom recognition, scalability strategies and incremental equality saturation. The results show that it outperforms several existing equality saturation engines, including the highly optimized egglog library. When used to reimplement an existing idiom recognition technique, the new design finds higher-quality idioms, 16× faster. Additionally, the design is able to natively express state-of-the-art custom equality saturation behavior such as incremental equality saturation and multi-phase rewriting strategies without any modification to the core library.

CCS Concepts: • **Software and its engineering** → **Compilers**; *Functional languages*; • **Theory of computation** → **Equational logic and rewriting**; • **Computing methodologies** → *Parallel algorithms.*

Keywords: Equality Saturation, Program Optimization, E-Graphs, Compiler Infrastructure, Rewrite Systems

CC '26, Sydney, NSW, Australia

ACM ISBN 979-8-4007-2274-5/2026/01
https://doi.org/10.1145/3771775.3786266

ACM Reference Format:
Jonathan Van der Cruysse, Abd-El-Aziz Zayed, Mai Jacob Peng, and Christophe Dubach. 2026. Parallel and Customizable Equality Saturation. In *Proceedings of the 35th ACM SIGPLAN International Conference on Compiler Construction (CC '26), January 31 – February 1, 2026, Sydney, NSW, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3771775.3786266

1 Introduction

Equality saturation (EqSat) [26] is a program optimization technique that represents many equivalent program variants in a finite data structure: an e-graph. EqSat enumerates the transformation space by applying rewrite rules until the e-graph is saturated. A cost model then extracts the best solution, solving the phase ordering problem. It has been widely used in many use-cases, from idiom recognition [29] to high-level synthesis [4].

Existing saturation engines [13, 21, 34, 37] have shown equality saturation scales to real-world applications. However, their implementation uses a single-threaded fixed saturation loop, offering little in terms of modularity or extensibility. The e-class analysis framework they expose also only supports lattice-based analysis and requires custom implementations for other metadata use cases [24]. This hinders the development of new equality saturation approaches.

This work addresses those limitations through three innovations: parallel e-matching and rewriting; declarative strategies; and generalized metadata. First, parallel e-matching is enabled using a thread-safe e-graph data structure and parallel rewriting is enabled by deferred updates in the form of commands. These commands can in parallel be generated, simplified, and scheduled for batch-parallel application, reducing time spent on e-graph updates.

Second, strategies replace fixed saturation loops with modular and extensible building blocks. Base strategies implement single rewrite steps consisting of e-matching and rewriting, then combinators declaratively construct more complex behavior. For example, the strategy below applies rules with exponential back-off scheduling until either saturation, a timeout, or an iteration limit is reached:

```
BackoffRuleApplication(rules, n, c)
  .withTimeout(timeout)
```

```
    .withIterationLimit(iterationLimit)
    .repeatUntilStable
```

Finally, generalized e-graph metadata provide a single abstraction that unifies the existing notion of e-class analyses with emerging use cases such as e-class versions [24]. This offers metadata as a built-in feature, whereas prior work have to fork an equality saturation engine to embed additional data directly in e-graph data structures.

These ideas are implemented in *Foresight*, an open-source equality saturation library [28]. While Foresight is implemented in Scala, the design and ideas presented could be implemented in any modern programming language.

Foresight is evaluated on four case studies: Horner's method and matrix associativity optimization, latent idiom recognition, existing saturation scalability mitigations, and incremental equality saturation. Horner's method and matrix associativity optimizations are benchmarks that permit comparison with recent work [36] on combining `egglog` [37] with MLIR [10]. Latent idiom recognition (LIAR) applies equality saturation to find idioms in functional programs [29]. The existing scalability mitigations are the custom algorithms for Isaria [27] and SymPy [7].

On the Horner's method and matrix associativity benchmarks, Foresight outperforms the `slotted`, `hegg` libraries; on matrix associativity it outperforms `egglog` thanks to its parallelism. The Foresight-based LIAR re-implementation obtains a geometric mean speedup of almost 16× over the original and produces superior solutions. The Isaria and SymPy strategies are encoded in 10 or fewer lines of code; expressing them using Foresight allows them to be applied in other settings, resulting in the finding that SymPy's strategy improves LIAR solutions while Isaria's degrades them.

To summarize, this paper contributes the following:

- A parallel e-matching and rewriting technique based on deferred commands (Section 4).
- A declarative strategy approach that generalizes the saturation loop and controls rule scheduling, stopping conditions, metadata, and more (Section 5).
- A generalization of e-graph metadata (Section 6).
- An empirical evaluation of Foresight on four case studies: two standard benchmarks, one established EqSat application, two known scalability mitigations, and incremental EqSat (Section 7).

2 Background: Saturation Engine Design

This section discusses the fundamentals of equality saturation, the existing landscape of equality saturation libraries, and the limitations of these libraries.

2.1 Equality Saturation

Equality saturation is a compiler technique that addresses the phase-ordering problem by deferring rewrite decisions. Rather than applying transformations sequentially, equality saturation adds all applicable rewrites to an *e-graph*, a data structure representing many equivalent program variants, until saturation. A cost heuristic then extracts the optimal variant from the saturated e-graph.

Figure 1 illustrates this workflow. First, an input arithmetic expression ① is turned into an e-graph ②, which wraps each operation (a variable reference, two constants, `*`, `pow`) from the expression in an *e-node*. Each e-node consists of an operator and arguments in the form of *e-classes*, groups of equivalent e-nodes. In the initial e-graph, each e-class, represented by the shading around e-nodes, has one e-node.

Through a process called *e-matching*, the e-graph ② is then searched for all occurrences of the left-hand sides of the two rewrite rules in ③. One such occurrence is found per rule; both are encoded in ④ as *matches*. Each match names the matched rule, the root e-class at which the match occurs, and a substitution that maps each symbol in the rewrite rule to an e-class in the e-graph.

Subsequently, each match is applied. Application starts by instantiating the right-hand side of the matches' rules with the match substitution mappings, yielding expressions that references e-classes: `d * pow(d, e - 1)` for the `"pow n"` rule match and `c + c` for `"mul 2"`. Each expression is then inserted in the e-graph (reusing identical e-nodes if already present) and its root is added to the match's root e-class in an operation called *union*.

Applying both matches from ④ results in the updated e-graph ⑤. This e-graph contains two e-classes containing two e-nodes each: `a` and `c`. Each e-node represents a different way to compute the same result, leading e-graph ⑤ to compactly encode six variants of the original program:

```
1 2 * pow(v, 3)
2 2 * v * pow(v, 2)
3 pow(v, 3) + pow(v, 3)
4 v * pow(v, 2) + pow(v, 3)
5 pow(v, 3) + v * pow(v, 2)
6 v * pow(v, 2) + v * pow(v, 2)
```

Classic equality saturation consists of repeating this match-and-apply process to further enrich the e-graph until it saturates or until a user-specified stopping criterion is reached. At that point, an *extractor* has a holistic view of the program's transformation space and uses that view to select a concrete program variant, usually based on a cost function. In this case, the extractor chooses expression ⑥.

2.2 Equality Saturation Engines

While many equality saturation applications build their own infrastructure in languages such as Java [26], Racket [18] and Scala [9, 29], the trend is to rely on a tool or library [13, 21, 34, 37]. Such libraries typically include five main components:

- An **e-graph implementation**.
- A single-threaded **rewrite engine** that implements and relies on an e-matching algorithm under the hood.

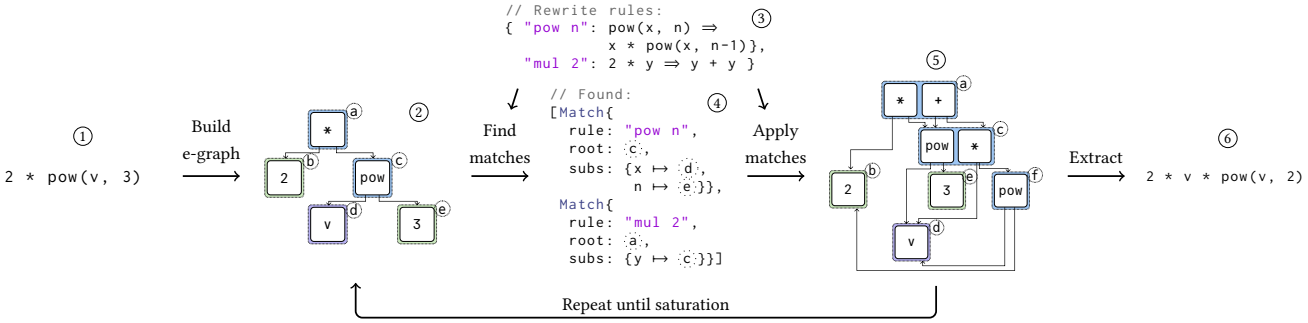


Figure 1. Equality saturation example. Input expression ① is encoded as e-graph ②. The e-graph is searched for occurrences of rewrite rules ③, leading to matches ④. Match application produces e-graph ⑤, from which expression ⑥ is extracted.

- An **e-class analysis** system that computes a domain-specific fact for each e-class in the e-graph.
- An **extractor** that selects expressions from an e-graph based on an application-specific cost function.
- An **equality saturation loop** that orchestrates rewriting, analyses and extraction.

The two most popular implementations of this model are egg [34] and `egglog` [37]. egg is an equality saturation library whereas `egglog` is an extension of Datalog with first-class support for e-graphs and equality saturation. Both are implemented in Rust, and `egglog` has Python bindings [22]. Additionally, `hegg` [13] is a Haskell alternative to egg.

Beyond these mature libraries, research projects such as `slotted` offer leaner egg-like libraries that experiment with novel features. In `slotted`'s case, that feature is slotted e-graphs [21], which encode variable bindings in e-graphs.

2.3 Limitations of Prior Saturation Engine Designs

As pre-implemented libraries enabled a range of modern equality saturation applications, these applications' techniques to overcome the rapid e-graph growth associated with classic equality saturation have revealed three limitations of state of the art libraries: single-threaded execution, limited support for custom saturation strategies, and limited metadata support.

Single-Threaded Rewriting. The e-graph implementations in current equality saturation libraries are based on a union-find data structure with path compression. This data structure optimizes for sequential performance but path compression is not thread safe, including for read-only operations. Moreover, e-graph insertions and unions translate to updates to data structures with complex invariants, creating another structural barrier to parallelization. As a result, equality saturation libraries are single-threaded.

Rigid Saturation Loops. Without parallelism to mask blow-ups in search space size, applications rely on curbing e-graph growth directly through custom *strategies*—algorithms that augment or replace the saturation loop.

A thread of general-purpose strategies is to apply Reinforcement Learning (RL) to guide rule application, where policy-based approaches learn which rewrites to apply [1, 2, 23]. An alternative to RL is search-based planning, which uses Monte Carlo Tree Search (MCTS) to explore alternative construction paths before committing [8].

Beyond learned policies and MCTS, many systems curb blow-ups with pragmatic, task-tuned controls layered on the vanilla loop. Typical ingredients include:

- **hard budgets**: timeouts, node caps [16, 17, 19, 26, 31];
- **phase separation** that staggers rule families (*e.g.,* simplification before expansion) [9, 27, 35];
- **rebasing**, where the system periodically extracts a best expression and reseeds the e-graph with only that expression to prune dead expansions [7, 9, 27];
- **rule scheduling** with exponential backoff to prevent hot rules from starving others [7];
- **sampling** of matches to cap per-iteration work [33];
- **e-graph segmentation** optimizes slices of the e-graph in isolation [32].

Some of these features, such as rule scheduling and hard budgets, benefit from built-in support in equality saturation libraries such as egg, yet the prevailing pattern is for user code to work around the library's saturation loop with ad-hoc, hard-coded strategies. This exposes a clear need for first-class, programmable saturation strategies in libraries.

Limited Metadata Support. Emerging schemes to reduce explosive e-graph growth, such as incremental equality saturation [24], rely on extending the e-graph with additional data, such as e-class versions. While equality saturation libraries support e-class analyses, they do not support generalized metadata, resulting in forks of equality saturation libraries that change the core e-graph data structure to support application-specific needs [24].

3 Overview

The remainder of this paper presents the key ideas behind Foresight, an equality saturation library in Scala that directly addresses the limitations highlighted above. These ideas are:

- **Parallel, deferred rewriting.** Foresight creates parallelism at both the e-matching and match rewriting stages of the rewriting process. E-Matching is parallelized by opting for a thread-safe e-graph data structure that can be safely searched from multiple threads. Match rewriting is parallelized rephrasing the task as the computation of deferred commands that are optimized before applying them in bulk to the e-graph.
- **Flexible strategies.** Instead of exposing a monolithic equality saturation engine, Foresight provides modular saturation strategies. Built-in base strategies implement rule application methodologies and are choreographed using combinators that describe saturation loops, stopping conditions, encapsulated analyses, and more. These strategies allow users to express complex saturation approaches with just a few lines of code.
- **Generalized metadata.** Foresight formulates metadata, including analysis and on-the-fly extraction results, as a stateful observer of the e-graph. E-Class analyses and analysis-based tree extraction are implemented as library-provided metadata instances.

These contributions let Foresight offer familiar functionality such as e-matching, e-class analyses and extraction, while turning the three previously-identified limitations of today's libraries—single-threaded execution, limited metadata, and ad-hoc growth-curbing tactics—into first-class capabilities: parallel rewriting, generalized metadata, and programmable strategies. The following sections each detail one of these ideas.

4 Parallel, Deferred Rewriting

Many aspects of the rewriting process in equality saturation involve computing tasks that have no dependencies on each other. Despite this, existing libraries operate in a serial fashion and prioritize single-threaded performance. This section describes how Foresight structures the rewriting process into phases to maximize per-phase parallelism.

4.1 Parallel E-Matching

As shown in Figures 1 and 2, each saturation iteration begins by finding rule matches over the e-graph. This process is inherently parallel: each rewrite rule can be checked independently against each e-class, and matching is semantically read-only.

However, in existing equality saturation libraries [13, 21, 34, 37], parallel matching is unsafe due to the use of union-find with path compression [25]. Although conceptually a lookup, the `find` operation of the union-find may mutate parent pointers during path compression. As matching frequently invokes `find` while traversing the e-graph, unsynchronized parallel reads can corrupt the union-find structure, making the e-graph unsafe for concurrent access.

In contrast, Foresight employs a thread-safe union-find in which parent pointers are stored in atomically updated arrays indexed by e-class identifiers. Path compression is preserved, but all parent updates are published using atomic operations, ensuring that `find` performs no unsafe writes. This design enables fully parallel e-matching over a shared e-graph without global locks.

4.2 Parallel Rewriting Using Deferred Commands

Once the list of matches is obtained, existing tools will sequentially apply each match, updating the e-graph with each application (Figure 2a, Line 12). To apply a match, the right-hand side of the matching rule is first reconstructed using the match's substitutions. For example, consider the rule

```
"pow n": pow(x, n) ⇒ x * pow(x, n-1)
```

One of the matches found for that rule in Figure 1 is:

```
Match{rule: "pow n", root: c,
      subs: {x ↦ d, n ↦ e}}
```

Applying this match reconstructs the right-hand side by substituting `x` with `d` and `n` with `e`, yielding the expression `d * pow(d, e - 1)`. This new term is then inserted into the e-graph as additional e-nodes and e-classes, connected through a union to the match's root e-class `c`.

Updating the e-graph is a sequential process as each e-node insertion either adds a new e-class to the e-graph or returns an existing e-class if the node is already in the graph. This condition creates a sequential dependency between match applications in prior approaches.

Unions are more flexible, and egg has found it beneficial to defer e-graph invariant maintenance [34]. First, it applies all matches, inserting e-nodes and recording unions, which may temporarily break e-graph invariants. Once all matches have been applied, egg rebuilds these invariants.

Foresight takes this idea one step further by deferring match applications. Instead of updating the e-graph immediately, each match generates *deferred commands* that record the necessary insertions and unions (Figure 2b, Line 3). The advantage of deferring match application is that the generation of commands is arbitrarily parallelizable across matches.

4.3 Command Simplification and Batching

Commands are encoded in Static Single Assignment (SSA) Form [6] over two operators: e-node insertions (`add`) and e-class unions (`union`). For the example expression `d * pow(d, e - 1)` from the previous section, along with the other example match application `c + c` from Figure 1, the sequence of commands corresponds to Figure 3a.

The advantages of commands are twofold: commands are simplified in parallel to reduce sequential application work

```
def rewrite_sequential(egraph, rules):
  matches = []
  for r in rules:
    for ecls in egraph.classes:
      case egraph.matchAndCompress(r, ecls) of
        Some(Match(data)):
          matches.append(Match(data))
        None:
          continue

  for match in matches:
    egraph.apply(match)
```

(a) Classic saturation process. `Highlighted` methods mutate e-graph data and must be executed sequentially.

```
def rewrite_parallel(egraph, rules):
  matches = findMatches(rules, egraph.classes)
  commands = computeUpdates(matches)
  addCmds, unionCmds = simplifyAndBatch(
    commands
  )

  return egraph.addMany(addCmds)
               .unionMany(unionCmds)
```

(b) Foresight's deferred saturation algorithm. `Highlighted` operations are fully parallel over their inputs. The `addMany` and `unionMany` methods are also partially parallelized.

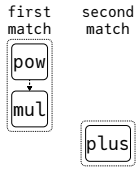
Figure 2. Pseudo-code comparison of the rewriting algorithms. By default, equality saturation performs rewrites until fixpoint.

```
// first match
%0 = add (2)
%1 = add (pow d, %0)
%2 = add (mul d, %1)
union %2, c

// second match
%3 = add (plus c, c)
union %3, a
```

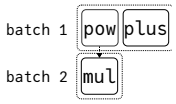
(a) Initial commands.

```
// first match
// (2) already in b
%1 = add (pow d, b)
%2 = add (mul d, %1)
union %2, c

// second match
%3 = add (plus c, c)
union %3, a
```

(b) After simplification.



(c) Dependencies for `add` commands.



(d) After batching.

```
// batch 1
%1 = add (pow d, b)
%3 = add (plus c, c)
// batch 2
%2 = add (mul d, %1)

union %2, c
union %3, a
```

(e) Reordered commands.

Figure 3. Command reordering from the matches in Figure 1.

and they are reordered into batches that allow for metadata update parallelism. Simplification recognizes that certain e-node additions and e-class unions are already in the e-graph. For example, `%0 = add (2)` inserts an integer literal `2` into the e-graph, but this literal is already present. Updating the instruction list from a parallel thread to pre-resolve this insertion, as in Figure 3b, reduces the number of sequential operations that will be applied to the e-graph.

Command reordering and batching forms maximal batches of `add` commands such that there are no dependencies between commands within a batch. For example, the dependency graph for the simplified commands (Figure 3c) is divided into the two batches from Figure 3d: one inserts `pow` and `plus` nodes; the other inserts a `mul` node. The result of reordering and batching is shown in Figure 3e.

Once created, each batch is applied to yield an updated version of the e-graph.

5 Saturation Strategies

Equality saturation is traditionally implemented as a library-defined loop that rewrites until convergence or a budget is exceeded. Foresight generalizes this with *strategies*: composable building blocks for rewriting behaviors and termination. This section shows how standard loops can be re-expressed.

5.1 From Loops to Strategies

The classic saturation loop, shown in Figure 4a, repeats until a fixpoint is reached. Since this loop never terminates for many applications, prior equality saturation tools offer a built-in saturation function (Figure 4b) that generalizes the above to include stopping conditions and rule scheduling [13, 34]. The stopping conditions are limited by the library to hard budgets such as timeouts, node caps or iteration caps; rule schedulers are customizable but are typically either maximal rule application (apply each rule during each rewriting iteration) or exponential backoff (each rule fires up to a quota of matches, then cools down for several rounds; both quota and cooldown expand after each cycle).

Foresight allows for more flexibility in how saturation is choreographed by introducing *saturation strategies*. A strategy declaratively describes the process of how to saturate an e-graph. These declarations come in two flavors:

- **base strategies** specify the rewriting algorithm; and
- **combinators** build more complex strategies from simpler building blocks.

The classic saturation loop, re-expressed as an Foresight strategy in Figure 4c, is a single base strategy (maximal rule application) followed by a combinator that runs it until fixpoint. A more typical instantiation of a `saturate` call would translate to Figure 4d. This strategy uses rewriting

```
def saturate_naive(egraph, rules):
  while true:
    egraph' = rewrite(egraph, rules)
    if egraph' == egraph:
      return egraph'
```

(a) Classic saturation loop.

```
def saturate(egraph, rules, condition, scheduler):
  i = 0
  while true:
    egraph' = rewrite(egraph, scheduler(rules, i))
    if egraph' == egraph or condition(egraph):
      return egraph'
    i += 1
```

(b) Saturation with a scheduler and stopping condition.

```
strategy = MaximalRuleApplication(rules)
  .repeatUntilStable
strategy(egraph)
```

(c) Figure 4a as an Foresight strategy.

```
strategy = BackoffRuleApplication(rules, n, c)
  .withTimeout(timeout)
  .withIterationLimit(iterationLimit)
  .repeatUntilStable
strategy(egraph)
```

(d) Figure 4b with backoff scheduler, as an Foresight strategy.

Figure 4. Existing saturation behaviors and how they are expressed as strategies in Foresight.

```
def withIterationLimit(inner_strat, limit):
  i = 0
  def apply(egraph):
    i += 1
    if i < limit:
      return inner_strat(egraph)
    return None
  return apply
```

Figure 5. Pseudocode implementation of `withIterationLimit`.

with a backoff scheduler and installs a timeout and iteration limit. Each of these limits becomes an additional combinator, chained to the previous strategy.

5.2 Core Interface

Foresight comes with a rich set of predefined strategies. Its base strategies implement maximal rule application, backoff scheduling, and stochastic application of matches. Built-in combinators compose strategies; repeat until fixpoint; install resource budgets; perform rebasing; augment e-graphs with additional analyses and metadata; and attach logging hooks.

A strategy is conceptually a higher order function that takes optional configuration parameters and returns a function of type `EGraph → EGraph | None`. Each strategy returns a function that either yields an updated e-graph or `None`. Returning `None` signals that a fixed point or stopping condition has been reached. Combinator strategies are configured by passing an inner strategy to apply. For example, Figure 5 implements the `withIterationLimit` combinator by extending another strategy with a counter. As syntactic sugar, the strategy examples in this paper use a dot syntax to simplify chaining, such that `foo(a, ...rest)` is equivalent to `a.foo(...rest)`.

Once constructed, a top-level strategy is applied to an e-graph by invoking it with its initial data, which is the default unless other data is specific. Other data only appears during strategy execution as a consequence of repetition combinators.

6 Generalized Metadata

Prior work supports metadata in the form of e-class analyses [34]. These analyses compute auxiliary information, such as constant folding results or type information, for each e-class. E-Class analyses operate on values that form a *lattice* [5], a common structure in static program analysis that provides a `join` function to merge information from multiple sources. A new value from a lattice is constructed with each e-node insertion, then values are propagated upward through the e-graph via the `join` function.

While effective for many applications, this design ties metadata to a single lattice structure. As a result, composing multiple analyses becomes cumbersome and customization beyond the lattice itself becomes impossible.

Foresight generalizes metadata support by allowing arbitrary, extensible annotations on the e-graph. Concretely, metadata in Foresight is formulated as a stateful observer via a callback mechanism that responds to the results of e-node insertions and e-class unions:

```
interface Metadata:
  def onAddMany(added: Seq[(ENode, EClass)],
                after: EGraph) -> Metadata
  def onUnionMany(equivalences: Set[Set[EClass
    ]],
                  after: EGraph) -> Metadata
```

The two observer callbacks encapsulate how metadata reacts to structural updates in the e-graph.

- `onAddMany` is invoked after a batch of e-node insertions has been applied. Metadata is computed as a function of the updated e-graph, *e.g.,* to initialize annotations for fresh e-classes.
- `onUnionMany` is invoked after a batch of e-class unions has been materialized. This hook reconciles metadata across merged e-classes, *e.g.,* by combining e-class analysis facts.

When multiple types of metadata are attached to the same e-graph, these callbacks are invoked in parallel for each type

of metadata. The rest of this section shows how Foresight's generalized metadata seamlessly supports both standard e-class analyses and *e-class versions*, a use case that standard analyses cannot express.

6.1 Analyses as Metadata

Foresight supports e-class analyses by formulating them as metadata, mapping e-classes to lattice values. Full implementation details are provided in the supplementary material.

The `onAddMany` callback initializes lattice values in parallel using **`make`**, whenever new nodes are inserted. The `onUnionMany` callback reconciles facts when e-classes are merged: it queries the e-graph's union-find for the leader e-class of the merged classes, combines all incoming facts with the **`join`** user-defined function, and then propagates the updated information upward.

Together, these two callbacks encapsulate e-class analysis updates, maintaining the analysis automatically as the e-graph evolves. Encoding analyses this way separates concerns: instead of interleaving analysis updates into the e-graph invariant maintenance algorithm, as prior approaches have done [34], expressing analyses as metadata simplifies analysis updates to an encapsulated metadata update after each e-graph update.

6.2 E-Class Versions as Metadata

Certain information cannot be represented as e-class analyses. For instance, recent work on incremental equality saturation relies on *e-class versions* to restrict equality saturation to the latest term added to an e-class [24]. These versions consist of one integer per e-class, identifying the term that was being processed when an e-class was inserted. Versions are not a function of e-node structure, nor do they propagate through the e-graph, making them ill-suited for e-class analyses and leading the designers of incremental equality saturation to fork egg to embed versions in e-classes.

In contrast, Foresight's metadata elegantly encodes e-class versions as metadata, with implementation details provided in the supplementary material. The `onAddMany` implementation sets the version of newly-inserted e-classes (typically inserted through rewriting rather than user action) to the version of the latest term being processed. `onUnionMany` defines the version of a union of e-classes as the minimum of those e-classes' versions but does not propagate this information upwards. Finally, a method `onAddNewTerm` allows user code to signal that a new term has been inserted into the e-graph, resulting in a new version that is assigned to all e-classes in the term.

7 Case Studies and Evaluation

This section describes four case studies that evaluate different aspects of Foresight's design. The first case study, Horner's method and matrix associativity, are established benchmarks [36] and that verify Foresight's correctness against existing tools. The next case study, latent idiom recognition, reimplements LIAR, a complex EqSat-based system in Scala [29]. This enables a comparison on a real-life use case with an EqSat implementation in the same programming language. A third case study demonstrates how Foresight's strategies significantly reduce the engineering effort of expressing custom saturation loops found in the literature [7, 27]. The final use case implements incremental equality saturation [24] to demonstrate generalized metadata.

Table 1. Memory usage comparison of the Java heap size used by Foresight and peak resident set size used by egg. Foresight is run in parallel with 8 threads.

Kernel	Foresight Heap Size	egg RSS
Horner	40 MB	4 MB
20mm	17 MB	3 MB
40mm	21 MB	4 MB
80mm	55 MB	13 MB

7.1 Experimental Setup

Foresight is an open-source Scala library that implements the ideas presented in this paper in addition to slotted e-graphs [21]. Foresight supports both Scala 2 and 3. Case studies 1, 2 and 5 use Scala 3.4.1 for its metaprogramming facilities; the remainder use Scala 2.12.19 for LIAR infrastructure compatibility.

All compilation is conducted on an Ubuntu 24.04 system with an Intel Xeon Gold 6254 CPU processor. Compiled benchmarks are run on a desktop machine (Intel Core i7-12700K) to reflect typical user setups. Unless otherwise stated, each experiment uses 8 threads for Foresight. The experimental setup for the comparisons against other libraries uses `egg` [34] version 0.10.0, `slotted` [21] version 0.0.35, and `hegg` [13] version 0.5.0.0. For `egglog` comparisons, the latest development version available at the time of writing is used (commit `ef90b97`). The matrix associativity benchmarks uses `egglog-experimental` (commit `908c47d`) for its support of dynamic cost functions.

7.2 Horner's Method and Matrix Associativity

This section confirms Foresight's correctness by applying it to classical optimization benchmarks: Horner's method and matrix associativity. These have previously been demonstrated [36] to be effective equality saturation library tests.

Applying Horner's method optimizes the computational complexity of polynomial evaluation by eliminating exponentiation operations and minimizing multiplications by rewriting it in the nested form shown below. While the naive approach requires $n(n+1)/2$ multiplications and n additions, applying Horner's method reduces this to only n multiplications and n additions.

$$P(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1} + a_n x^n$$
$$P(x) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n)))$$

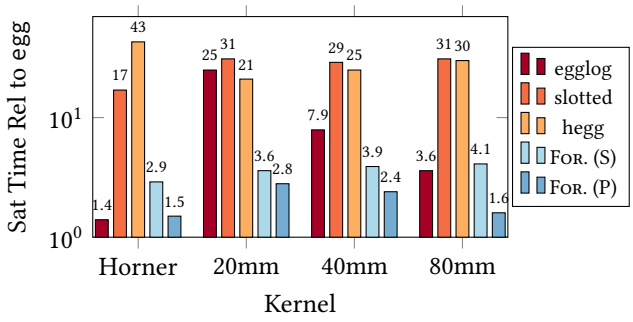


Figure 6. Comparison of saturation time across many implementations of equality saturation. For. (S) and For. (P) refer to sequential and parallel Foresight. The baseline is egg.

Horner's method is derived through eight rewrite rules: commutative and associative properties to rearrange terms, distributive properties to factor common subexpressions, a recursive exponentiation rule to decompose powers, and algebraic identities for simplification. The cost function prioritizes multiplication over exponential operations, and addition over multiplication operations.

Foresight discovers Horner's method as expected and Figure 6 shows the saturation times compared to other libraries, normalized against the egg library baseline. Across all benchmarks, parallel Foresight shows faster saturation than the sequential Foresight, `hegg` and `slotted` implementations. This performance advantage stems from parallel e-matching where the search for all eight rules can proceed concurrently across the e-graph, a particularly beneficial optimization given that the saturated e-graph contains 1260 e-nodes. Foresight's effectiveness becomes even more pronounced as saturation approaches the fixed point. At this stage, most right-hand sides of rewrites are already in the e-graph, rendering subsequent rewrite applications no-ops. Foresight's command simplification exploits this convergence pattern and reduces the sequential work required during command application, explaining the performance gains observed. egg consistently outperforms Foresight in absolute saturation time on these benchmarks; the difference is explained by implementation-level factors. In particular, Foresight's object-heavy JVM implementation uses significantly more memory than egg's Rust implementation (Table 1), whereas algorithmically Foresight's only additional memory use is for storing deferred commands.

Similarly, the associativity of matrix multiplication is used to find the optimal association of a chain of matrix multiplications. This optimization requires only two rewrite rules—the associative property—and a cost function that measures the number of scalar multiplications. Foresight finds the optimal association for the tested chains of 20, 40 and 80 matrix multiplications and Figure 6 shows the saturation times. Both sequential and parallel Foresight outperform the highly-optimized `egglog` library on `20mm` and `40m`; on `80mm`, Foresight's parallelism is the deciding factor in speeding past `egglog`. Slowdowns relative to `egg` are again explained by the increased memory use in Table 1.

Table 2. BLAS idioms where Foresight finds a better solution than LIAR. In all other cases, the solutions are identical. Benchmark run time speedups (*Sp.*) vary from 1× to 10×.

Kernel	LIAR solution	Foresight solution	Sp.
2mm	3 × axpy + 2 × gemv + 4 × memset	2 × gemm + 1 × memset + 1 × transpose	3×
gemm	3 × axpy + 1 × gemv + 3 × memset	1 × gemm + 1 × transpose	10×
slim-2mm	1 × gemm + 1 × gemv + 2 × memset + 1 × transpose	2 × gemm + 2 × memset + 2 × transpose	1×

7.3 Latent Idiom Recognition

To evaluate Foresight on a larger application, this section reimplements LIAR [29], a technique that recognizes array idioms using equality saturation. As both implementations are in Scala, this case study demonstrates the advantages of Foresight when compared to the existing Scala landscape and shows how Foresight integrates into the larger SHIR compiler infrastructure [20] on which LIAR is based.

Using Foresight, we rebuilt LIAR with significantly less effort, leveraging slotted e-graphs and high-level rewrite APIs. Slotted e-graphs eliminate the need for LIAR's original De Bruijn machinery, enabling variable binding to be expressed declaratively and managed directly in the e-graph structure. LIAR's type inference, originally implemented as a built-in component of its equality saturation engine, is implemented using Foresight as an e-class analysis. Additionally, LIAR e-graphs are equipped with an extraction e-class analysis to support an extraction-based beta-reduction rule [9, 34]. Foresight has built-in support for extraction analyses and its rewriting engine runs analyses in parallel.

Two experiments evaluate the Foresight reimplementation of LIAR: 1. replicating LIAR's idiom recognition evaluation targeting BLAS libraries using the same set of PolyBench benchmark kernels, and 2. rerunning the same evaluation with varying thread counts to examine the scaling behavior of Foresight's parallel architecture.

7.3.1 Idiom Quality and Saturation Speed.

Table 2 and Figure 7 summarize the results of the first experiment. Foresight exceeds the idiom recognition capabilities of the original LIAR engine and is faster.

Given the same number of saturation iterations, Foresight identifies equally or more idiomatic solutions than LIAR. For example, in `2mm`, the baseline emits a composition of `axpy`, `dot`, `gemv`, and `memset` calls, while Foresight's result is a concise sequence of `gemm`, `memset`, and `transpose`, reflecting higher-quality idiom recognition that results in

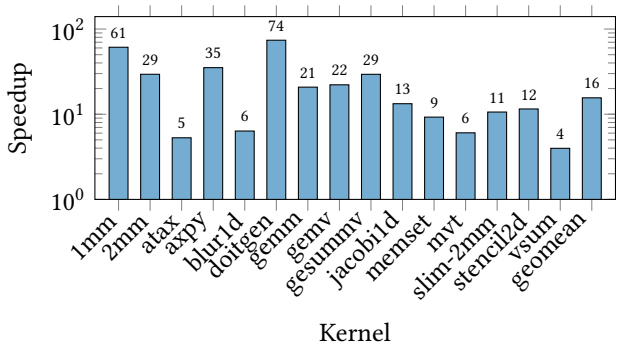


Figure 7. Saturation speedups of Foresight relative to the original LIAR engine across BLAS benchmarks.

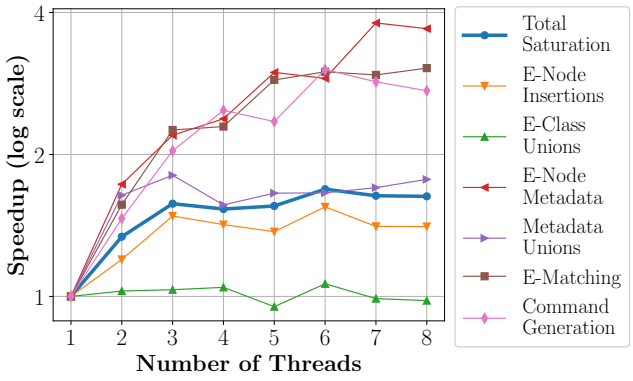


Figure 8. Parallel speedup across individual components of Foresight's equality saturation engine for the stencil2d kernel. Rule matching and application scale effectively. Metadata updates scale partially, while hashconsing stays sequential.

a 3× run time speedup on the `2mm` benchmark and 10× on `gemm`. The new `slim-2mm` solution is performance-neutral.

These higher-quality results are also found faster: Figure 7 shows that Foresight achieves a geometric mean total saturation speedup of almost 16×. Total speedups vary across benchmarks due to differences in e-graph size as slotted e-graphs expose additional equivalences and eliminate De Bruijn shift rules, which results in both smaller or larger graphs depending on the benchmark. Speedups nonetheless remain consistently strong, indicating a more efficient rewrite engine regardless of graph size.

7.3.2 Parallel Scalability. Figure 8 presents results from the second experiment, in which the BLAS evaluation is rerun with varying thread counts to assess internal scalability. Speedup is reported for several core saturation operations.

Rule matching and rule application scale near-linearly, benefiting from deferred, batched updates and thread-safe graphs. Metadata propagation via `addMany` and `unionMany` also benefits from parallelism, though with different scaling characteristics. The `addMany` metadata step continues to improve at higher thread counts due to parallelization across both metadata types and individual e-node additions. In contrast, the `unionMany` metadata step plateaus once the number of threads exceeds the number of active metadata analyses, restricting further speedup.

The overall saturation speedup for the LIAR `stencil2d` benchmark levels off at 1.7×, reflecting the interplay between the scalable e-matching, command generation and e-node metadata components, the moderately parallelizable metadata union phase, and the sequential nature of hashcons updates. At higher thread counts, these sequential components dominate: at eight threads, e-class unification is 30% of total time. This explains the plateau in total throughput.

These results confirm that Foresight's architecture enables efficient and fine-grained parallelism for most rewrite and analysis tasks, with scalability bounded by the intrinsic parallelism available in each phase. Results are application-specific; `stencil2d` obtains a speedup of 1.7×, while `80mm` from the previous section obtains a 2.6× speedup.

7.4 Existing Scalability Strategies (Applied to LIAR)

This section demonstrates the flexibility of Foresight's composable strategy system by showing how to express two saturation strategies from prior work: the phased pipeline from *Isaria* [27] and a multi-round EqSat schedule that was previously applied to symbolic regression [7, 14]. Where both original systems are built around monolithic EqSat runners, the equivalent Foresight implementations are each 10 or fewer lines of readable code.

7.4.1 Isaria (phased + prune-and-rebase). Isaria is an equality saturation–based auto-vectorization framework that has three types of rewrite rules: 1. expansion rules, which turn smaller expressions into larger expressions; 2. simplification rules, which turn larger expressions into smaller ones; and 3. vectorization rules, which replace expressions with vector instructions. To keep equality saturation tractable, its saturation strategy is to repeatedly interleave expansion, simplification, and pruning of intermediate search states. This pruning is done via *rebasing*, which replaces the e-graph with a single extracted expression. After repeated expansion, simplification and rebasing, Isaria applies a single pass of vectorization rules. Intuitively, expansion–simplification–vectorization phases separate exploration from exploitation; rebasing curbs e-graph growth.

In Foresight, we capture Isaria's strategy with per-phase budgets, fixed-point loops, and periodic rebasing.

```
def phase(rules: Seq[Rule]):
  MaximalRuleApplication(rules)
    .withTimeout(phaseTimeout)
    .repeatUntilStable
phase(expansionRules)
  .thenApply(phase(simplificationRules))
  .withTimeout(expansionAndSimplTimeout)
  .thenRebase(extractor, areEquivalent)
  .repeatUntilStable
  .thenApply(phase(idiomRules))
```

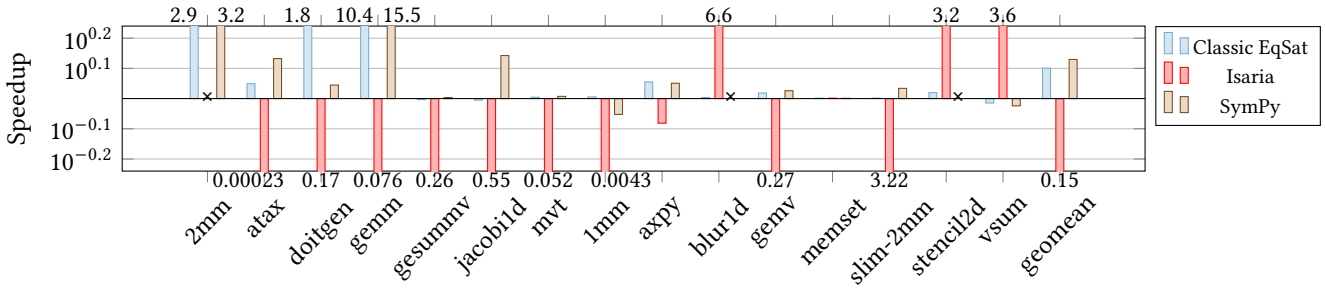


Figure 9. Run time speedup of Foresight's solutions compared to baseline LIAR solutions. Each bar represents the ratio of the baseline run time and the Foresight solution run time. Higher is better.

The original Isaria driver reports 819 lines of Rust code (excluding whitespace/comments) [27], whereas the port above uses a handful of composable combinators whose parameters expose the same operational levers.

7.4.2 SymPy (bounded + backoff + multi-round).

De França and Kronberger [7] use equality saturation to reduce redundant fitted parameters in symbolic regression and report consistently better results than SymPy's simplifier. Below, we encode their bounded, conservatively scheduled equality saturation as a first-class Foresight strategy with per-iteration budgets, cooldown-style backoff, periodic extraction, and multiple rounds of saturation. For brevity, we refer to this strategy SymPy, but it denotes the EqSat-based bounded-scheduler pipeline that was compared to SymPy's simplifier [7, 14].

```
BackoffRuleApplication(rules, n, cool0ff)
  .withIterationLimit(iterationLimit)
  .repeatUntilStable
  .thenRebase(extractor)
  .withIterationLimit(cycles)
  .repeatUntilStable
```

7.4.3 Scalability Strategy Results.

Figure 9 shows how the Isaria and SymPy strategies carry over to LIAR. The plot reports benchmark run times relative to the original LIAR baseline. Foresight with a standard EqSat loop consistently matches or outperforms the baseline. SymPy excels on atax, gemm, jacobi1d, and slim-2mm, but times out (slower than 60s) on blur1d and stencil2d. Conversely, Isaria does well on blur1d and stencil2d but performs poorly on other kernels.

7.5 Incremental Equality Saturation

This section shows that generalized metadata allows Foresight to support use cases not covered by existing equality saturation libraries. This is demonstrated by implementing incremental equality saturation, based on the e-class version metadata from Section 6.2.

Incremental equality saturation [24] is a recent technique that efficiently optimizes a sequence of input expressions. This efficiency is due to reusing a single evolving e-graph across optimization runs, extending the structure-sharing benefits of equality saturation temporally. Each e-class in that e-graph is tagged with a version number, and rewrites are applied only to e-classes relevant to the current version, avoiding redundant exploration while remembering previously-discovered facts. Existing equality saturation libraries do not have a mechanism to support e-class versions. Previous experiments with incremental equality saturation hence depended on a specialized fork of `egg`.

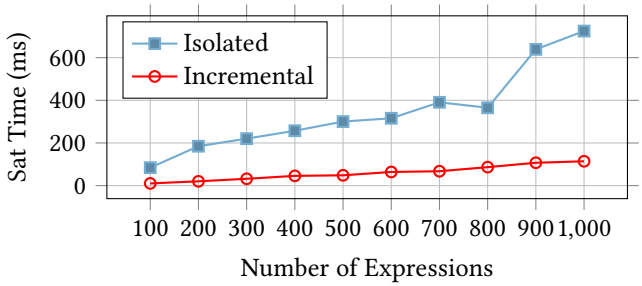


Figure 10. Saturation time of incremental vs standard EqSat. Inputs are identical; saturation uses one thread.

Foresight's metadata allows for e-class versions to be implemented without library modifications, as described in Section 6.2. Leveraging e-class versions to implement incremental equality saturation, Figure 10 reproduces the expected behavior by applying incremental saturation to increasingly long sequences of randomly generated polynomial expressions. Saturating these expressions incrementally is much more efficient than saturating them individually, indicating that the Foresight-based incremental saturation implementation satisfactorily reuses previous knowledge.

8 Related Work

Existing equality saturation libraries such as `egg` [34] and `egglog` [37] are finely tuned frameworks suitable for practical equality saturation applications. Other implementations target specific use cases: `hegg` [13] provides equality saturation support for Haskell, while `slotted` [21] adds native support for variable bindings. Foresight's e-graphs are also slotted, simplifying binders. Design decisions in the aforementioned libraries limit parallelism and inhibit customizable saturation techniques, which has led numerous

projects to fork these libraries or layer custom code on top of them [7, 24, 27]. Foresight's strategies and metadata suffice to express this custom behavior without library changes.

E-morphic [3] parallelizes extraction using simulated annealing, which is useful when rewriting is cheap and extraction is expensive. This is orthogonal to Foresight's parallel rewriting, which is useful when rewriting is expensive.

Relational E-Matching [38] reformulates E-Matching as relational queries. `egglog` and `hegg` use relational e-matching; `egg` and Foresight use standard e-matching [15]. Relational e-matching is orthogonal to the parallel search and deferred updates in this work.

Constable [32] segments e-graphs into slices that are processed independently. This scalability mitigation is orthogonal to the ideas presented in this work and could be implemented as a scalability strategy in future work.

A flurry of recent techniques explore integrating equality saturation with larger systems, *e.g.,* Julia and MLIR [11, 12, 32, 36]. While integration with production compilers is not the primary focus of this work, Section 7.3 demonstrates that its contributions enable customizable, extensible, and performant integration with existing compiler infrastructure.

9 Conclusion

This paper has presented Foresight, an equality saturation library that prioritizes flexibility and parallelism. Foresight aims to enable innovation in equality saturation research by exposing a set of composable building blocks (flexible saturation strategies and generalized metadata) for implementing custom saturation techniques. These building blocks naturally express recent approaches such as incremental equality saturation [24] and multi-phase, rebased saturation [7, 27].

Despite this focus on extensibility, Foresight consistently outperforms `hegg` and `slotted`, which are also implemented in a high-level language and also support slotted e-graphs, respectively. When parallelism is enabled, Foresight leverages its deferred approach to rewriting to outperform `egglog` on workloads that benefit from concurrent rewriting. The highly-optimized `egg` engine remains faster than Foresight across all evaluated benchmarks, reflecting its emphasis on single-threaded performance and minimal memory use.

Looking forward, Foresight's metadata system provides a promising foundation for proof reconstruction. By recording the provenance of e-class merges, such as whether they arise from congruence closure or from specific rewrite rule applications, metadata can encode the information needed to reconstruct equational proofs after saturation.

10 Data Availability Statement

The software and data that support the findings of this paper are released within Zenodo [30] under a CC-BY 4.0 license.

Acknowledgments

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grants Program [grant RGPIN-2020-05889], the Canada CIFAR AI Chairs Program, and doctoral training scholarships 304858 and 370894 from the Fonds de recherche du Québec.

References

[1] George-Octavian Barbulescu. 2023. An Inquiry into Database Query Optimisation with Equality Saturation and Reinforcement Learning. *Diss. University of Cambridge* (2023).

[2] George-Octavian Bărbulescu, Taiyi Wang, Zak Singh, and Eiko Yoneki. 2024. Learned graph rewriting with equality saturation: A new paradigm in relational query rewrite and beyond. *arXiv preprint arXiv:2407.12794* (2024).

[3] Chen Chen, Guangyu Hu, Cunxi Yu, Yuzhe Ma, and Hongce Zhang. 2025. E-morphic: Scalable Equality Saturation for Structural Exploration in Logic Synthesis. *arXiv preprint arXiv:2504.11574* (2025).

[4] Jianyi Cheng, Samuel Coward, Lorenzo Chelini, Rafael Barbalho, and Theo Drane. 2024. SEER: Super-Optimization Explorer for High-Level Synthesis using E-graph Rewriting. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 1029–1044. doi:10.1145/3620665.3640392

[5] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 238–252.

[6] Ronald Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490. doi:10.1145/115372.115320

[7] Fabrício Olivetti de França and Gabriel Kronberger. 2023. Reducing Overparameterization of Symbolic Regression Models with Equality Saturation. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '23)*. ACM, New York, NY, USA, 1064–1072. doi:10.1145/3583131.3590346

[8] 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*. 40–52.

[9] Thomas Koehler, Andrés Goens, Siddharth Bhat, Tobias Grosser, Phil Trinder, and Michel Steuwer. 2024. Guided Equality Saturation. *Proc. ACM Program. Lang.* 8, POPL, Article 58 (Jan. 2024), 32 pages. doi:10.1145/3632900

[10] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. doi:10.1109/CGO51591.2021.9370308

[11] 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

[12] Jules Merckx, Alexandre Lopoukhine, Samuel Coward, Jianyi Cheng, Bjorn De Sutter, and Tobias Grosser. 2025. eqsat: An Equality Saturation Dialect for Non-destructive Rewriting. arXiv:2505.09363 [cs.PL] https://arxiv.org/abs/2505.09363

[13] Rodrigo Mesquita. 2022. hegg: Equality Saturation in Haskell. https://github.com/alt-romes/hegg. Accessed: 2025-08-20.

[14] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. 2017. SymPy: symbolic computing in Python. *PeerJ Computer Science* 3 (2017), e103. doi:10.7717/peerj-cs.103

[15] Leonardo Moura and Nikolaj Bjørner. 2007. Efficient E-Matching for SMT Solvers. In *Proceedings of the 21st International Conference on Automated Deduction: Automated Deduction* (Bremen, Germany) *(CADE-21)*. Springer-Verlag, Berlin, Heidelberg, 183–198. doi:10.1007/978-3-540-73595-3_13

[16] 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

[17] Anjali Pal, Brett Saiki, Ryan Tjoa, Cynthia Richey, Amy Zhu, Oliver Flatt, Max Willsey, Zachary Tatlock, and Chandrakana Nandi. 2023. Equality Saturation Theory Exploration à la Carte. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 258 (Oct. 2023), 29 pages. doi:10.1145/3622834

[18] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 1–11. doi:10.1145/2737924.2737959

[19] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 1–11. doi:10.1145/2737924.2737959

[20] Christof Schlaak, Tzung-Han Juang, and Christophe Dubach. 2022. Memory-Aware Functional IR for Higher-Level Synthesis of Accelerators. *ACM Trans. Archit. Code Optim.* 19, 2, Article 16 (Jan. 2022), 26 pages. doi:10.1145/3501768

[21] Rudi Schneider, Marcus Rossel, Amir Shaikhha, Andrés Goens, Thomas Kœhler, and Michel Steuwer. 2025. Slotted E-Graphs. In *Proceedings of the ACM on Programming Languages, PLDI 2025*, Vol. 9. ACM.

[22] Saul Shanabrook. 2025. egglog-python: Python bindings for the egglog Rust library. https://github.com/egraphs-good/egglog-python.

[23] Zak Singh. 2022. Deep reinforcement learning for equality saturation. *MPhil Research Project. University of Cambridge, June* (2022).

[24] Rupanshu Soi, Benjamin Driscoll, Ke Wang, and Alex Aiken. 2025. Incremental Equality Saturation. In *EGRAPHS Workshop at ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2025)*. Seoul, South Korea. https://rupanshusoi.github.io/pdfs/egraphs-25.pdf

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

[26] 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

[27] Samuel Thomas and James Bornholt. 2024. Automatic Generation of Vectorizing Compilers for Customizable Digital Signal Processors. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 19–34. doi:10.1145/3617232.3624873

[28] Jonathan Van der Cruysse. 2025. Foresight: A Scala Equality Saturation Library. https://github.com/jonathanvdc/foresight.

[29] Jonathan Van der Cruysse and Christophe Dubach. 2024. Latent Idiom Recognition for a Minimalist Functional Array Language Using Equality Saturation. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 270–282. doi:10.1109/CGO57630.2024.10444879

[30] Jonathan Van der Cruysse, Abd-El-Aziz Zayed, Mai Jacob Peng, and Christophe Dubach. 2025. *Parallel and Customizable Equality Saturation*. doi:10.5281/zenodo.17955956

[31] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 2021. Vectorization for digital signal processors via equality saturation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 874–886. doi:10.1145/3445814.3446707

[32] Arya Vohra, Leo Seojun Lee, Jakub Bachurski, Oleksandr Zinenko, Phitchaya Mangpo Phothilimthana, Albert Cohen, and William S. Moses. 2025. Mind the Abstraction Gap: Bringing Equality Saturation to Real-World ML Compilers. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 284 (Oct. 2025), 28 pages. doi:10.1145/3763062

[33] 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. *Proc. VLDB Endow.* 13, 12 (July 2020), 1919–1932. doi:10.14778/3407790.3407799

[34] 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 (1 2021), 29 pages. doi:10.1145/3434304

[35] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 255–268. https://proceedings.mlsys.org/paper_files/paper/2021/file/cc427d934a7f6c0663e5923f49eba531-Paper.pdf

[36] 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

[37] 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 (June 2023), 25 pages. doi:10.1145/3591239

[38] Yihong Zhang, Yisu Remy Wang, Max Willsey, and Zachary Tatlock. 2022. Relational e-matching. *Proc. ACM Program. Lang.* 6, POPL, Article 35 (Jan. 2022), 22 pages. doi:10.1145/3498696

Received 2025-11-11; accepted 2025-12-10