Guided Equality Saturation

THOMAS KŒHLER, Inria, France and ICube lab, Université de Strasbourg, CNRS, France
ANDRÉS GOENS, University of Amsterdam, Netherlands and University of Edinburgh, UK
SIDDHARTH BHAT, University of Edinburgh, UK
TOBIAS GROSSER, University of Cambridge, UK and University of Edinburgh, UK
PHIL TRINDER, University of Glasgow, UK
MICHEL STEUWER, Technische Universität Berlin, Germany and University of Edinburgh, UK

Rewriting is a principled term transformation technique with uses across theorem proving and compilation. In theorem proving, each rewrite is a proof step; in compilation, rewrites optimize a program term. While developing rewrite sequences manually is possible, this process does not scale to larger rewrite sequences. Automated rewriting techniques, like greedy simplification or equality saturation, work well without requiring human input. Yet, they do not scale to large search spaces, limiting the complexity of tasks where automated rewriting is effective, and meaning that just a small increase in term size or rewrite length may result in failure.

This paper proposes a semi-automatic rewriting technique as a means to scale rewriting by allowing human insight at key decision points. Specifically, we propose *guided equality saturation* that embraces human guidance when fully automated equality saturation does not scale. The rewriting is split into two simpler automatic equality saturation steps: from the original term to a human-provided intermediate *guide*, and from the guide to the target. Complex rewriting tasks may require multiple guides, resulting in a sequence of equality saturation steps. A guide can be a complete term, or a *sketch* containing undefined elements that are instantiated by the equality saturation search. Such sketches may be far more concise than complete terms.

We demonstrate the generality and effectiveness of guided equality saturation using two case studies. First, we integrate guided equality saturation in the Lean 4 proof assistant. Proofs are written in the style of textbook proof sketches, as a series of calculations omitting details and skipping steps. These proofs conclude in less than a second instead of minutes when compared to unguided equality saturation, and can find complex proofs that previously had to be done manually. Second, in the compiler of the RISE array language, where unguided equality saturation fails to perform optimizations within an hour and using 60 GB of memory, guided equality saturation performs the same optimizations with at most 3 guides, within seconds using less than 1 GB memory.

CCS Concepts: • **Theory of computation** → **Equational logic and rewriting**; **Automated reasoning**; • **Software and its engineering** → **Compilers**; • **General and reference** → **Performance**.

Additional Key Words and Phrases: e-graphs, equality saturation, theorem provers, optimizing compilers

Authors' addresses: Thomas Kœhler, Inria, France and ICube lab, Université de Strasbourg, CNRS, France, thomas.koehler@inria.fr; Andrés Goens, University of Amsterdam, Netherlands and University of Edinburgh, UK, a.goens@uva.nl; Siddharth Bhat, University of Edinburgh, Scotland, UK, siddharth.bhat@ed.ac.uk; Tobias Grosser, University of Cambridge, UK and University of Edinburgh, UK, tobias.grosser@cst.cam.ac.uk; Phil Trinder, University of Glasgow, Scotland, UK, phil.trinder@glasgow.ac.uk; Michel Steuwer, Technische Universität Berlin, Germany and University of Edinburgh, Scotland, UK, michel.steuwer@tu-berlin.de.

1 INTRODUCTION

Term rewriting [Baader and Nipkow 1998] provides compositional reasoning and optimizations through a formal theory of syntactic changes and is used from theorem proving [Bachmair and Ganzinger 1994; Hsiang et al. 1992] to compilation [Dershowitz 1993; Hagedorn et al. 2020; Visser et al. 1998]. Automatic theorem provers such as Z3 provide optimized algorithms for equational reasoning [de Moura and Bjørner 2007], and the Glasgow Haskell Compiler [Jones et al. 2001] optimizes a widely-used functional programming language through term rewriting. In both cases, the definition and exploration of the rewriting search space is automatic and scales (for specific domains) to complex problems. Yet, term rewriting offers benefits even for domains or problem sizes where decision procedures do not exist or do not scale well, but where humans succeed in identifying desirable rewrite sequences.

For example, proving some theorems requires specific insights that are hard to come up with automatically. Manually specifying rewrites allows injecting such insights, and conveying a particular chain of reasoning steps. As an illustration, consider the following proof that inversion in groups is an involution: for a group G and an element $a \in G$, we claim that $(a^{-1})^{-1} = a$. The proof is conceptually straightforward and can be found in any standard textbook:

$$(a^{-1})^{-1} \xrightarrow{\text{mul. one}} (a^{-1})^{-1} \cdot 1 \xrightarrow{\text{mul. inverse}} (a^{-1})^{-1} \cdot (a^{-1} \cdot a) \xrightarrow{\text{mul. assoc.}} ((a^{-1})^{-1} \cdot a^{-1}) \cdot a \xrightarrow{\text{mul. inverse}} 1 \cdot a \xrightarrow{\text{mul. one}} a$$

This proof shows how the different group axioms imply this result in a couple of reasoning steps, using the multiplicative identity of one, the defining property of inverse and associativity of multiplication at different points. Interestingly, this proof requires a creative insight highlighted in red: how did we know it was a good idea, *"out of nowhere"*, to multiply the starting term by $a^{-1} \cdot a$? The axiom of the multiplicative inverse is universally quantified, $\forall x \in G, x^{-1} \cdot x = 1$, so there's a potentially infinite number of ways to instantiate it. The insight is crucial for the proof, and automated term rewriting systems struggle to produce this proof. In fact, automatically finding these insights on arbitrary rewrite systems is impossible: word problems on such equational theories are known to be undecidable in general [Evans 1978].

Controlling rewrites manually also has value in optimizing compilers. Strategy languages [Visser et al. 1998] empower programmers to manually control when to apply individual rewrite rules and compose them step-by-step into complex compiler optimizations. ELEVATE [Hagedorn et al. 2020] shows that manually defined rewriting strategies perform *complex optimizations* of functional array programs in less than a second of compilation time, producing high performance code. However, these complex optimizations emerge from thousands of rewrite steps making the rewriting strategies challenging to write manually.

Automated rewrite techniques are appealing as they promise to release humans from the burden of specifying when and where rewrite rules are applied, e.g. to prove a theorem or optimize a program. This is particularly important for more complex applications and domains where manually rewriting can be quite difficult and quickly becomes tedious for anything but the simplest of cases.

There are many applications for which existing automatic rewrite techniques work very well. These are, for example, well-behaved applications where the rewrite system has a convergence property, i.e., confluence and termination. A concrete example is the group axioms extended with certain additional identities, like the involutive property of group inverses above, which can be obtained by a method called Knuth-Bendix completion [Knuth and Bendix 1983]. Similarly, many practical but simple compiler optimizations can be achieved with simple greedy rewriting.

Unfortunately, greedy rewriting gets stuck in local optima. Equality saturation [Tate et al. 2009; Willsey et al. 2021] and the related Congruence Closure [Nelson and Oppen 1980; Nieuwenhuis and Oliveras 2007] avoid this problem by using a graph data structure to efficiently represent and

rewrite many equivalent terms. Aggressively exploring many rewrites in this way is expensive, and in some cases the graph grows exponentially, quickly exceeding practical memory limits.

We observe that when reasoning about rewrites informally, humans tend to skip simple steps but discuss intermediate points. A proof like the one above for groups might be written like that for didactic reasons in an introductory course, but usually, only the key insight would be shown. One might write, e.g. $(a^{-1})^{-1} = (a^{-1})^{-1} \cdot (a^{-1} \cdot a) = a$. This is a proof sketch, skipping some steps and omitting details like the concrete rewrites, but the details left out are much simpler to reconstruct than the key insight. Similarly, when discussing program transformations, we think in intermediate steps and sketch out how the transformed terms roughly ought to look, without writing out entire programs. So we may seek to tile a loop, without specifying exactly how its body changes.

Based on this insight this paper proposes *guided equality saturation*, a novel semi-automatic rewriting technique that factors complex rewrite problems into a sequence of simpler rewrite problems, each sufficiently simple to be found by equality saturation. The expert controls rewriting by specifying a sequence of *guides* that provide human insight. If problems are too complex, additional guides make them feasible. Thus, guided equality saturation gradually scales to increasingly complex problems, as we will see in our two case studies.

Our *theorem proving case study* demonstrates that guided equality saturation allows more proofs to be found significantly faster, compared to unguided techniques that sometimes fail to find a proof at all. We show that human-added guides closely resemble the reasoning steps found in textbooks.

Our *program optimization case study* shows that guided equality saturation allows more optimizations to be found with significantly lower computational cost, compared to unguided equality saturation that fails to find optimizations by exceeding runtime or memory limits. We demonstrate how human experts can write guides as incomplete term *sketches*, greatly simplifying the specification of guides. The sketch guide is instantiated to a complete term by the equality saturation.

To summarize, the main contributions of this paper are:

- *Guided equality saturation*, a semi-automated technique offering a novel, practical trade-off between manual rewriting and automated rewriting. Human-provided *guides* break a single infeasible rewrite problem into a sequence of feasible rewrite problems, each solvable with equality saturation. Guides can either be specified as concrete intermediate terms, or in many cases as imprecise *sketches* to simplify their specification.
- A *theorem proving case study* integrating guided equality saturation in the Lean 4 theorem prover [de Moura and Ullrich 2021] and demonstrating proofs from group theory and ring theory as examples. The insights injected as guides either make proofs up to two orders of magnitude faster, or make proofs feasible where fully automated techniques fail.
- A *program optimization case study* using guided equality saturation to optimize programs in the functional array language RISE [Steuwer et al. 2022]. Seven advanced optimizations of a matrix multiplication application are implemented, including loop blocking, vectorization, and multithreading. Unguided equality saturation fails to perform the five most complex optimizations, even given an hour and 60 GB of RAM. Using at most three sketch guides, each 10 times smaller than the complete program, all optimizations are applied in seconds using less than 1 GB of RAM.

We start in Section 2 with some background on equality saturation, the automated technique we build on, before introducing guided equality saturation in Section 3. We present our case studies in Sections 4 and 5, before discussing related work in Section 6. We discuss how to use guides in Section 7, and conclude in Section 8.

2 STRENGTHS AND WEAKNESSES OF EQUALITY SATURATION

This section provides some background on equality saturation, the automated technique we build on. We demonstrate its strengths compared with greedy rewriting, a popular automatic technique used, for example, in GHC [Jones et al. 2001] and LLVM [Lattner and Adve 2004]. We then show equality saturation's weaknesses by discussing examples that are too challenging for it.

2.1 Greedy Rewriting and the Phase Ordering Problem

Greedy rewriting applies rewrites greedily, aiming to minimize a cost function in every local rewrite step. Let us look at a practical example related to optimizing functional array code. We want to fuse operators to avoid writing intermediate results to memory, such as for this program:

$$(map \; (map \; f)) \circ (transpose \circ (map \; (map \; g))) \qquad \longmapsto^* \qquad \text{(A)}$$
$$(map \; (map \; (f \circ g))) \circ transpose \qquad \qquad \text{(B)}$$

The initial program (A) applies function g to each element of a two-dimensional matrix using two nested maps, transposes the result, then applies function f to each element. The optimized program (B) avoids an intermediate matrix, transposing the input before applying g and f to each element. (B) can be derived from (A) by applying the following rewrite rules in the correct order:

$$transpose \circ map \; (map \; a) \longmapsto map \; (map \; a) \circ transpose \qquad \text{(1)}$$
$$a \circ (b \circ c) \longmapsto (a \circ b) \circ c \qquad \text{(2)}$$
$$map \; a \circ map \; b \longmapsto map \; (a \circ b) \qquad \text{(3)}$$

To perform greedy rewriting, we have to choose a cost function. As minimizing term size results in maximizing the number of fused operators, it is a reasonable choice. If we greedily apply only rewrite rules that decrease the term size, we will only apply rule (3) as this is the only rule that reduces term size (from 5 to 4 terms). However, rule (3) cannot be directly applied to term (A): it is a local optimum. The only way to reduce term size further is to first apply the other rewrite rules, which may or may not pay off depending on future rewrites. We observe a similar phenomenon for the theorem proving example in the introduction, where we first have to increase the term size before we can reduce it. But in greedy rewriting, we do not apply rules that do not make *local* progress towards our cost function.

This makes it easy to get stuck in a local optimum, and, therefore, does not work well for applications where the global optimum is significantly better than local optima. The challenge is that often the global benefit of applying a rewrite rule depends on future rewrites. In the compiler community, the problem of automatically deciding when to apply each rewrite rule is known as the *phase ordering problem* [Touati and Barthou 2006].

2.2 Mitigating the Phase Ordering Problem with Equality Saturation

Different communities have developed techniques to mitigate the phase ordering problem, and related problems of finding sequences of rewrites, by automatically exploring many possible ways to apply rewrite rules. The theorem proving community have developed Congruence Closure [Nelson and Oppen 1980; Nieuwenhuis and Oliveras 2007], after which the program optimization community developed the closely-related equality saturation technique [Tate et al. 2009; Willsey et al. 2021] that we focus on in this paper.

2.2.1 Equality Saturation. Starting from an input term, equality saturation grows an equality graph (*e-graph*) by applying all possible rewrites iteratively until reaching a fixed point (saturation), achieving a performance goal, or timing out. An e-graph efficiently represents a large set of equivalent terms and is grown by repeatedly applying rewrite rules in a purely additive way.
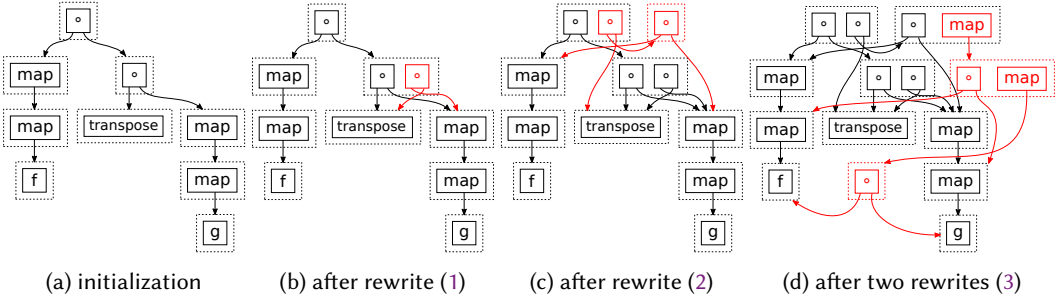
| (a) initialization | (b) after rewrite (1) | (c) after rewrite (2) | (d) after two rewrites (3) |

Fig. 1. Growing an e-graph for the term $(map\ (map\ f)) \circ (transpose \circ (map\ (map\ g)))$. An e-graph is a set of e-classes themselves containing equivalent e-nodes. The dashed boxes are e-classes, and the solid boxes are e-nodes. New e-nodes and e-classes are shown in red.

Instead of replacing the matched left-hand side of a rewrite rule with its right-hand side, the equality between both sides is recorded in the e-graph. After growing the e-graph, the best program found is extracted from it using a cost model, e.g. one that selects the fastest program.

2.2.2 Exploring Past the Local Optimum with Equality Saturation. The e-graph (equivalence graph) is used to efficiently represent and rewrite a set of equivalent terms, intuitively:

- An *e-graph* is a set of equivalence classes (e-classes)
- An *e-class* is a set of equivalent nodes (e-nodes)
- An *e-node* $F(e_1, .., e_n)$ is an n-ary function symbol (F) from the term language, associated with child e-classes (e_i)

Figure 1a shows the e-graph for the program (A) from Section 2.1. E-classes are shown as boxes with a dashed outline and e-nodes are shown as boxes with a solid outline. The e-class children of an e-node are shown as directed edges forming the e-graph.

To start the exploration phase, the initial e-graph is iteratively grown by applying rewrite rules non-destructively (Figures 1b to 1d). On each equality saturation iteration, all possible rewrites are applied in a breadth-first manner. This contrasts with standard term rewriting where a single possible rewrite is selected in a depth-first manner, requiring careful ordering of rewrite rule applications. In Figure 1, we only apply a handful of rewrite rules per iteration for the sake of simplicity. Rewrite rule applications are considered even if they do not locally lower cost, which avoids getting stuck in local optima. When applying a rewrite rule, the equality between its matched left-hand side and its instantiated right-hand side is recorded in the e-graph. In contrast, standard term rewriting destructively replaces the matched left-hand side with the instantiated right-hand side, producing a new term from the initial one.

Crucially for efficiency, an e-graph is far more compact than a naive set of terms, as equivalent sub-terms are shared. E-graphs can represent exponentially many terms in polynomial space, and even infinitely many terms in the presence of cycles [Willsey et al. 2021]. To maximize sharing, a *congruence invariant* is maintained: intuitively identical e-nodes should not be in different e-classes (Figure 2). Later we will see that even extensive sharing does not necessarily prevent e-graph sizes from growing exponentially or exceeding practical memory limits.

2.2.3 Extracting a Global Optimum with Equality Saturation. The exploration phase terminates, and rewrite rules stop being applied, when a fixed point is reached (saturation), or when another stopping criteria is reached (e.g. timeout or a certain goal has been achieved). If saturation is reached, it means that all possible rewrites have been explored. After the exploration of rewrites
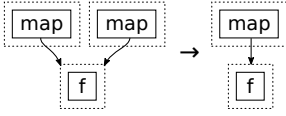
Fig. 2. The congruence invariant simplifies the e-graph on the left by merging two identical e-nodes for *map f* into a single e-node as shown on the right.
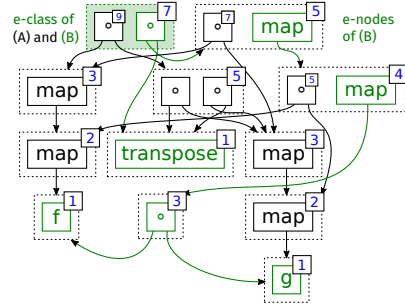

Fig. 3. Smallest term size computed during extraction, shown for each e-class in the top-right corner in blue. Where it differs from its e-class value, the smallest term size of e-nodes is also shown. The e-class and e-nodes of the smallest term (B) are shown in green.

has terminated, we extract a term from the e-graph according to a cost function, e.g., the one with the smallest term size. The extracted term is a global optimum if saturation was reached.

The *extraction* procedure can be a relatively simple bottom-up e-graph traversal, similar to Knuth [1977]'s generalization of Dijkstra's algorithm, if the cost function is local. Non-local cost functions require more complex extraction procedures [Wang et al. 2020; Wu et al. 2019].

A *local cost function c* can be defined as a function of a term language symbol F and the costs of its children, i.e. c has signature $c(F(k_1 : K, .., k_n : K)) : K$ with costs of type K. For example, term size is a local cost function: $\text{termSize}(F(k_1, .., k_n)) = 1 + \sum_i k_i$.

The term sizes computed during a bottom-up extraction procedure are shown in Figure 3. The figure reveals that the grown e-graph contains a smaller term of term size 7 in the same e-class as the original term (A) of term size 9 (top left in Figure 3). The extracted term is indeed program (B).

2.2.4 Successful Applications of Equality Saturation. The *map/transpose* example (A) demonstrates how equality saturation can succeed where greedy rewriting is not sufficient. This is also true for many applications in other domains. In theorem proving, while a proof for $a^{-1} \cdot (a \cdot b) = b$ can easily be found via greedy rewriting from the group axioms, a proof of $1^{-1} = 1$ is out of the reach of greedy rewriting but can be found with equality saturation [1]. We can think of this in analogy of overcoming local optima, by taking for cost function the "simplicity" of the term, as defined by the rewrites that will simplify the term (i.e., find a normal form if the rewrite system is Noetherian).

A renewed interest in equality saturation has been sparked by the recent egg library [Willsey et al. 2021] with applications in optimizing linear algebra [Wang et al. 2020], shrinking 3D CAD (Computer-Aided Design) models [Nandi et al. 2020], optimizing deep learning programs [Smith et al. 2021; Yang et al. 2021], vectorizing digital signal processing code [VanHattum et al. 2021], inferring rewrite rules [Nandi et al. 2021], and more.

The closely related congruence closure technique has been effectively used to produce proofs in proof assistants [Nieuwenhuis and Oliveras 2005; Selsam and de Moura 2016] and automated theorem provers [de Moura and Bjørner 2007].

---

[1]This, in fact, depends on the axiomatization of groups. We use the standard axiomatization from mathematics which requires inverses to be both left and right inverses, as well as identities to be both left and right identities. This can be relaxed to only require the left (or right) properties, but then e.g. groups being monoids becomes a non-trivial theorem and not direct by definition. With this axiomatization, equality saturation also fails to find the proof of $1^{-1} = 1$.

2.3 The Limits of Equality Saturation

While equality saturation has found many successful uses, it has limits. We have seen already in the introduction a theorem proving example that does not work with equality saturation, as crucial human insight is required for the proof. In this section, we explore an additional example highlighting a significant, practical scaling limitation of equality saturation. In this example, the e-graph grows extremely rapidly, adding many e-classes and e-nodes in each iteration without enough sharing opportunity to fit the e-graph within a practical memory limit.

2.3.1 Reaching the Limits of Equality Saturation. Loop tiling is a traditional compiler optimization that improves memory access patterns and with it cache performance. Tiling is typically performed on multiple nested loops. To demonstrate the scaling behavior of equality saturation, we first attempt to perform the tiling of a single loop, then two nested loops and finally three nested loops. As before, we perform the optimizations on functional array code, but now the first parameter of *map* represents the array size. We use the ∘-associativity and *map*-fusion rewrites rules (see (2) and (3) in Section 2.1) and two more rules below:

$$map \; n1 \; (map \; n2 \; f) \longmapsto transpose \circ (map \; n2 \; (map \; n1 \; f)) \circ transpose \qquad (4)$$

$$map \; (n1 \times n2) \; f \longmapsto join \circ (map \; n1 \; (map \; n2 \; f)) \circ (split \; n2) \qquad (5)$$

To perform one-dimensional tiling, a single loop (represented by the functional *map*) is split according to the rules into two nested loops, as shown here:

$$map \; (n1 \times 32) \; f \; \longmapsto^* \; join \circ (map \; n1 \; (map \; 32 \; f)) \circ (split \; 32) \qquad \text{(Tile1D)}$$

This is a trivial rewrite only applying rule (5). A more complex sequence of rewrites has to be performed to tile two nested loops into four:

$$map \; (n1 \times 32) \; (map \; (n2 \times 32) \; f) \qquad \longmapsto^* \qquad \text{(Tile2D)}$$
$$join \circ (map \; n1 \; (map \; 32 \; join)) \circ (map \; n1 \; transpose) \circ$$
$$(map \; n1 \; (map \; n2 \; (map \; 32 \; (map \; 32 \; f)))) \circ$$
$$(map \; n1 \; transpose) \circ (map \; n1 \; (map \; 32 \; (split \; 32))) \circ (split \; 32)$$

This is a bit more challenging, but still within easy reach of equality saturation. Finally, to tile three nested loops into six:

$$map \; (n1 \times 32) \; (map \; (n2 \times 32) \; (map \; (n3 \times 32) \; f)) \qquad \longmapsto^* \qquad \text{(Tile3D)}$$
$$(map \; (n1 \times 32) \; (map \; (n2 \times 32) \; join) \circ join) \circ join \circ$$
$$(map \; n1 \; transpose \circ (map \; n2 \; (map \; 32 \; transpose) \circ transpose)) \circ$$
$$(map \; n1 \; (map \; n2 \; (map \; n3 \; (map \; 32 \; (map \; 32 \; (map \; 32 \; f)))))) \circ$$
$$(map \; n1 \; (map \; n2 \; transpose)) \circ (map \; n1 \; (map \; n2 \; (map \; 32 \; transpose)) \circ transpose) \circ$$
$$(split \; 32) \circ (map \; (n1 \times 32) \; (map \; n2 \; (map \; 32 \; (split \; 32))) \circ (split \; 32))$$

Here equality saturation struggles and fails to perform the rewrite. Figure 4 shows the memory footprint required by equality saturation for performing 1D (4a), 2D (4b), and 3D (4c) tiling. The e-graph for three-dimensional tiling grows very quickly, requiring large amounts of memory before the rewrite completes. This points to a general characteristic of equality saturation: either a successful rewrite sequence is found relatively quickly, or, computational costs explode.

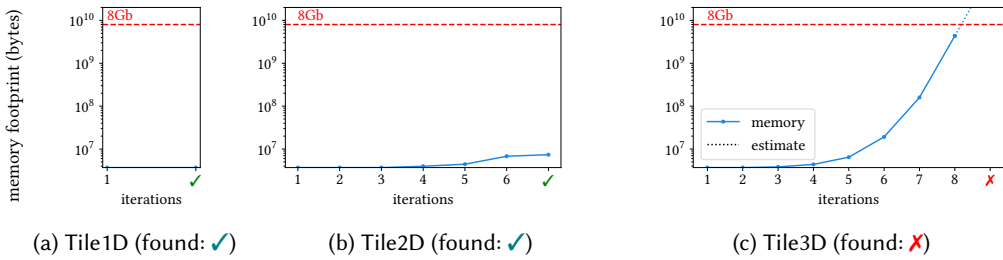(a) Tile1D (found: ✓)   (b) Tile2D (found: ✓)   (c) Tile3D (found: ✗)

Fig. 4. Memory footprint during equality saturation performing loop tiling. While one- and two-tiling is performed easily, three-dimensional loop tiling fails as the e-graph requires too much memory.

2.3.2 Ways to Reduce E-Graph Growth. Various ad-hoc ways to limit the growth of e-graphs have been discussed, such as limiting the number of rules applied [Wang et al. 2020; Willsey et al. 2021], which risks not finding rewrites that require an omitted rule. An application-specific alternative is to use an external solver to speculatively add equivalences [Nandi et al. 2020], but this requires the identification of sub-tasks that can benefit from being delegated. It is also possible to trade-off between the exploitation of greedy rewriting and the exploration of equality saturation [Kourta et al. 2022], but this requires a good enough heuristic cost function to make local decisions.

As an alternative mitigation to this issue, this paper proposes a novel semi-automatic approach: allowing experts to *guide* the rewrite process by breaking a single infeasible equality saturation process into a sequence of feasible equality saturations. On top of reducing computational cost, this also provides a mechanism for experts to inject insights into the rewrite process, supporting applications such as the theorem proving example from the introduction.

3 GUIDED EQUALITY SATURATION

This section introduces *guided equality saturation*. Figure 5 illustrates how a human expert guides the rewrite process towards a final goal by specifying a sequence of intermediate goals that we call *guides*. By doing so, they break a rewrite problem that is too complex for unguided equality saturation into simpler problems, each sufficiently simple to be solved by equality saturation.

We first present the algorithm for implementing guided equality saturation and demonstrate that it is capable of performing the three-dimensional loop tiling for which unguided equality saturation fails. Then we discuss how to use terms as guides and introduce *sketch guides* that simplify the specification of guides.
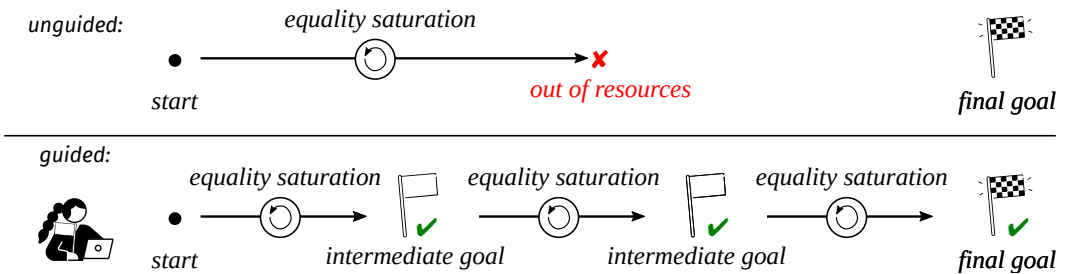

Fig. 5. Where unguided equality saturation (top) fails to reach its rewrite goal, *guided equality saturation* (bottom) succeeds. The human expert guides the rewrite process by providing intermediate goals (*guides*), and with them breaks the single infeasible equality saturation into a sequence of feasible equality saturations.

```
1  def guided_equality_saturation(term, eqsats):
2   for eqsat in eqsats:
3    egraph = eqsat.initial_egraph(term)
4    while not (eqsat.goal_is_reached(egraph) or egraph.is_saturated_or_timeout()):
5
6       # apply standard equality saturation as in egg
7       matches = []
8       for rw in eqsat.rewrites:
9         for (subst, eclass) in egraph.ematch(rw.lhs):
10          matches.append((rw, subst, eclass))
11       for (rw, subst, eclass) in matches:
12         eclass2 = egraph.add(rw.rhs.subst(subst))
13         egraph.merge(eclass, eclass2)
14       egraph.rebuild()
15
16    if eqsat.goal_is_reached(egraph):
17      term = eqsat.extract_term_satisfied_by_goal(egraph)
18    else:
19      fail("goal_not_found")
20   return term
```

Listing 1. Pseudocode of guided equality saturation, applying a sequence of standard equality saturations (line 2, Figure 6), each rewriting towards a new (intermediate) goal.
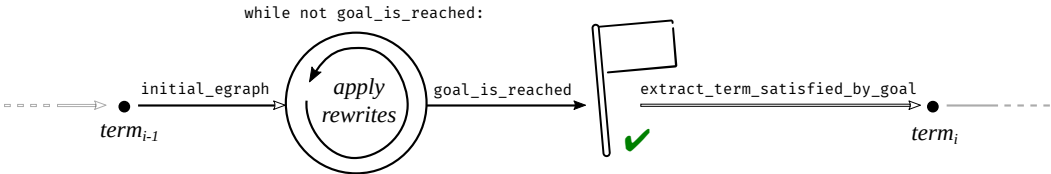

Fig. 6. Performing a single equality saturation as part of the overall guided equality saturation process. With the term from the prior equality saturation $term_{i-1}$ an e-graph is initialized (listing 1 line 3). As long as the next goal is not reached (while loop in listing 1 line 4) rewrites are applied to grow the e-graph as usual. Once the goal has been satisfied, $term_i$ satisfying the goal is extracted from the e-graph (listing 1 line 17) to start the next equality saturation aiming to reach the next goal.

3.1 Guided Equality Saturation Algorithm

Listing 1 shows the guided equality saturation algorithm as pseudocode in the style of the egg paper [Willsey et al. 2021]. A sequence of equality saturations is performed (line 2, Figure 6). Each equality saturation initializes a new e-graph (line 3) and performs standard equality saturation iterations as described by Willsey et al. [2021] (lines 7–14). Note, that each equality saturation can use a different set of rewrites. We will explore the practical implications of choosing different sets of rewrites for different equality saturations as part of our case studies.

After each equality saturation iteration, the algorithm checks whether the intermediate goal has been reached (line 4). If so a term satisfying the goal is extracted (line 16-17) and assigned to the term variable. This term will be used to initiate a new e-graph for the next equality saturation.

An equality saturation step may fail to find an intermediate goal, and if so the process has failed. Section 7 will discuss how users can deal with search failures.

Scaling Beyond Equality Saturation. Figure 7 shows the three-dimensional loop tiling example from Section 2.3.1. On the left, we see how unguided equality saturation runs out of memory quickly, while on the right how a single intermediate goal as a guide makes the rewrite feasible. Two equality saturations are now performed. The first finishes after 6 iterations reaching the intermediate goal, before the second equality saturation performs 7 further iterations growing a new e-graph. We will discuss guides shortly, and show the sketch guide used here in Section 3.3.

(a) Equality saturation (found: ✗)   (b) **Guided** equality saturation (found: ✓)
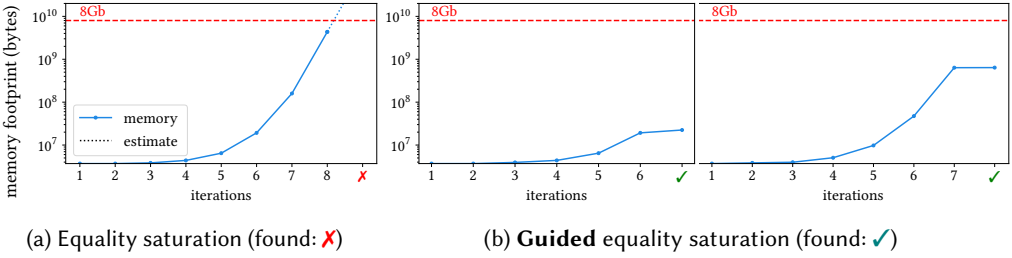
Fig. 7. Memory footprint for performing three-dimensional loop tiling for unguided equality saturation on the left and guided equality saturation on the right. Where unguided equality saturation runs out of memory, a single guide is sufficient for guided equality saturation to succeed. The guide is described in Section 3.3.

3.2 Terms as Goals

Goals, final or intermediate, may be specified either as concrete terms or as imprecise sketches. We first discuss uses of concrete terms, and postpone the discussion of sketches to Section 3.3.

When using a concrete term as a goal, the e-graph is initialized (Listing 1 line 3) with the start term *and* the goal term. This allows rewrites to be applied to both the start term and the goal term when searching for a rewrite sequence that establishes their equivalence. Checking that the goal is reached is straightforward, as we simply check if the goal and start term are in the same e-class.

Using terms as goals is useful in many situations, such as in theorem proving. As an example, let us analyze the process of equational reasoning about mathematical identities with an algorithmic lens. When reasoning about a mathematical term with a piece of paper, it is common to write a proof sketch as a sequence of concrete terms corresponding to a sequence of key reasoning steps. Mathematicians do this to incrementally explore potential proofs and record different attempts at reaching a goal term from an initial term, until successful. On paper, we usually skip simple intermediate steps and only document the key reasoning for non-obvious steps by annotating them with the theorem used. A good proof sketch should be enough to guide an informed reader so that they can fill in the reasoning gaps. Section 4 shows that using terms as intermediate goals replicates this form of reasoning closely by providing key reasoning steps as intermediate goals, and automates the detailed and routine reasoning between them.

3.3 Sketches as Goals

For some applications specifying a concrete intermediate term is tedious, as terms may be large and even a minor syntactic mistake results in the term not being found. Let us again consider the three-dimensional loop-tiling example. In Section 3.1 we showed that a single intermediate goal specified as a guide is sufficient to perform the optimization where unguided equality saturation fails. But how complex is it to write the guide?

An experienced performance engineer can easily come up with the intuition that performing loop-tiling can be broken into two steps: first, splitting the loops, and then reordering them. So specifying a guide where the three nested loops are split into six makes sense, but specifying the concrete term is overly tedious:

$$(map\ (n1 \times 32)\ (map\ (n2 \times 32)\ join) \circ join) \circ join \circ$$
$$(map\ n1\ (map\ 32\ (map\ n2\ (map\ 32\ (map\ n3\ (map\ 32\ f)))))) \circ \qquad \text{(Split3D Term Guide)}$$
$$(split\ 32) \circ (map\ (n1 \times 32)\ (map\ n2\ (map\ 32\ (split\ 32)))) \circ (split\ 32))$$

$$S ::= ? \mid F(S, .., S) \mid contains(S) \mid S \vee S$$

$$\mathcal{R}[\![?]\!] = T = \{F(t_1, .., t_n)\}$$
$$\mathcal{R}[\![F(s_1, .., s_n)]\!] = \{F(t_1, .., t_n) \mid t_i \in \mathcal{R}[\![s_i]\!]\}$$
$$\mathcal{R}[\![contains(s)]\!] = \mathcal{R}[\![s]\!] \cup \{F(t_1, .., t_n) \mid \exists t_i \in \mathcal{R}[\![contains(s)]\!]\}$$
$$\mathcal{R}[\![s_1 \vee s_2]\!] = \mathcal{R}[\![s_1]\!] \cup \mathcal{R}[\![s_2]\!]$$

Fig. 8. Grammar of SKETCHBASIC (top) and terms represented by SKETCHBASIC (bottom).

The engineer must not just specify the six nested loops (represented as maps) in the second line, but also the precise transformations of the input and output data. To greatly simplify the specification of guides for such applications, we introduce *sketches*. We provide an open-source implementation of sketches on top of the egg library, which includes this loop tiling example (https://github.com/Bastacyclop/egg-sketches).

Writing the same guide as a sketch is much simpler, and focuses on the key intuition that the desired term must contain six nested loops:

contains (*map n*1 (*map* 32 (*map n*2 (*map* 32 (*map n*3 (*map* 32 *f*))))))   (Split3D Sketch Guide)

Informal program snippets are often used by performance engineers to visualise and explain program optimizations. This can be observed in many papers that use rewriting strategies or schedules to specify optimizations, such as [Adams et al. 2019; Anderson et al. 2021; Chen et al. 2018; Ikarashi et al. 2021; Koehler and Steuwer 2021; Ragan-Kelley et al. 2013; Sioutas et al. 2020]. Our sketches can be seen as a formalization of these informal program snippets.

When using a sketch as a goal, the e-graph is initialized only with the starting term, as we cannot add a sketch to the e-graph. As we will see next, sketches are specified as logical predicates. To check if the sketch goal has been reached, we perform e-class analysis to efficiently check if there are terms in the e-graph that satisfy the sketch predicate. To obtain a term for the next equality saturation we perform a sketch-satisfying extraction that is described in Section 3.3.3. But, first, we formally define sketches.

3.3.1 Defining Sketches. Sketches are specified in a SKETCHBASIC language with just four constructors. The syntax of SKETCHBASIC and the set of terms that the constructors represent are defined in Figure 8. A sketch s represents a set of terms $\mathcal{R}[\![s]\!]$, such that $\mathcal{R}[\![s]\!] \subset T$ where T denotes all terms in the language we rewrite. We say that any $t \in \mathcal{R}[\![s]\!]$ satisfies sketch s.

The ? sketch is the least precise, representing all terms in the language. The $F(s_1, .., s_n)$ sketch represents all terms that match a specific n-ary function symbol F from the term language, and whose n children satisfy sketches s_i. The *contains*(s) sketch represents all terms containing a term that satisfies sketch s: the least solution to the recursive $\mathcal{R}[\![contains(s)]\!]$ equation. Finally, the $s_1 \vee s_2$ sketch represents terms satisfying either s_1 or s_2.

3.3.2 Sketch Precision. Writing a useful sketch to guide an optimization search requires striking a balance between being too precise and too vague. An overly precise sketch may exclude valid terms with a slightly different structure. An overly vague sketch may lead to finding undesirable terms. This balance also interacts with the set of rewrite rules used, since terms that may be found by the search are $\mathcal{R}[\![s]\!] \cap \mathcal{E}_{rules}[\![t]\!]$ where $\mathcal{E}_{rules}[\![t]\!]$ represents the set of terms that can be discovered to be equivalent to the initial term t according to the given *rules*. This means that using a more restricted set of rules generally enables specifying less precise sketches.

3.3.3 Sketch-Satisfying Extraction. As a sketch may be satisfied by many terms, we require a cost function to extract a single term from the e-graph that satisfies the sketch. The extracted term is used as the starting point for the next equality saturation search. More formally, to extract the best term that satisfies a sketch s from an e-class e of an e-graph g we define a helper function $ex(g, e, s, c)$, where c is a cost function that must be monotonic and local (Section 2.2.3). The function ex returns a cost K associated with the best term, and as extraction may fail, the return value is optional. Thus, the return type is $Option[(K, Term)]$. For efficiency ex is memoized. Then, extract uses ex to return a term if possible, failing otherwise:

$$\text{extract}(g, e, s, c) = t \quad \text{if} \quad ex(g, e, s, c) = Some\ (\_, t)$$

ex is recursively defined over the four SKETCHBASIC cases as follows.

Case 1: $ex(g, e, ?, c) = Some\ extract_{term}(g, e, c)$. This extracts a term that minimises c from the e-class e of the e-graph g (Section 2.2.3).

Case 2: $ex(g, e, F(s_1, .., s_n), c)$. The lowest cost term from all $F(e_1, .., e_n)$ e-nodes in e is selected.

$$ex(g, e, F(s_1, .., s_n), c) = foldl\ best\ None$$

$$\left\{ Some \left(\begin{array}{c} c(F(k_1, .., k_n)), \\ F(t_1, .., t_n) \end{array} \right) \middle| \begin{array}{c} F(e_1, .., e_n) \in e, \\ \forall i.\ ex(g, e_i, s_i, c) = Some\ (k_i, t_i) \end{array} \right\},$$

$$\text{where } best(d_1, d_2) = \begin{cases} \text{if } k_1 \leq k_2 \text{ then } d_1 \text{ else } d_2 & \forall i.\ d_i = Some\ (k_i, \_) \\ d_i & \exists i.\ d_i = Some\ (k_i, \_) \\ None & \text{otherwise} \end{cases}$$

Case 3: $ex(g, e, contains(s), c)$. The lowest cost term from the base and recursive cases is selected. The recursive cases correspond to $F(e_1, .., e_n)$ e-nodes in e where an e_j satisfies $contains(s)$.

$$ex(g, e, contains(s), c) = foldl\ best\ None\ candidates$$

$$\text{with } candidates = \{ex(g, e, s, c)\} \quad \cup$$

$$\left\{ Some \left(\begin{array}{c} c(F(k_1, .., k_j, .., k_n)), \\ F(t_1, .., t_j, .., t_n) \end{array} \right) \middle| \begin{array}{c} F(e_1, .., e_n) \in e \\ ex(g, e_j, contains(s), c) = Some\ (k_j, t_j) \\ \forall i.i \neq j, ex(g, e_i, ?, c) = Some\ (k_i, t_i) \end{array} \right\}$$

The implementation memoizes a dummy $ex(g, e, contains(s), c) = None$ result before evaluating $ex(g, e_j, contains(s), c)$ to avoid entering a cycle.

Case 4: $ex(g, e, s_1 \vee s_2, c) = best(ex(g, e, s_1, c), ex(g, e, s_2, c))$. The lowest cost alternative is selected.

The next two sections present two case studies applying guided equality saturation to theorem proving (Section 4) and program optimization (Section 5). Each provides distinct challenges for equality saturation and demonstrates how guidance provides a practical way to tackle them.

4 CASE STUDY: THEOREM PROVING

Machine-checked theorem proving has seen several milestone achievements in mathematics, like the Kepler conjecture [Hales et al. 2017] or the four-color theorem [Gonthier et al. 2008]. These, however, have been more sparse and not always at the cutting edge of research in mathematics. Of central importance to these machine-checked proofs are interactive theorem provers (ITPs). These provide an interactive environment that can guide a user step-by-step in constructing a proof. In particular, the Lean theorem prover [de Moura and Ullrich 2021] has garnered much attention from the Mathematics community. Recent applications have reached fields like number theory [Buzzard et al. 2020], addressing research-level questions in that space, e.g. [Scholze 2021].

Hence we use Lean 4 for this study. ITPs like Coq or Isabelle also have a long history of successful uses in computer science and verification, and could have been used in our study.

E-graphs and congruence closure have been used in ITPs with considerable success. The key idea is that an e-graph can be used to keep metadata about how it was constructed, enabling the construction of a proof witness: a sequence of rewrites that prove equivalence by transitivity [Corbineau 2006; Nieuwenhuis and Oliveras 2005; Selsam and de Moura 2016]. Many modern provers like Coq, Isabelle or Lean also feature rich sets of so-called tactics, meta-programs that enable partial proof automation by operating in the intermediate proof state. Both Coq and Isabelle have tactics that use congruence closure, as well as older versions of Lean.

4.1 Equational Reasoning in Lean

Consider the following snippet of Lean 4 code, which corresponds to the example from Section 1:

```
theorem inverse_involution [Group G] (g : G) : g⁻¹⁻¹ = g := by
rw [←mul_one g⁻¹⁻¹, ←inv_mul_self g, ←mul_assoc, inv_mul_self g⁻¹, one_mul]
```

The `rw` tactic allows us to write the proof as a series of rewrite steps, which are sequentially applied to the current goal. The ← character denotes the rewrite should be applied from right to left, and arguments like $g^{-1^{-1}}$ or g instantiate universally-quantified theorems to concrete rewrites. The rewrites are specified by names, while the intermediate states are implicit. However, most ITPs provide tactics to make this calculation more explicit; in Lean this tactic is called `calc`. With it, the proof looks more like in the introduction:

```
g⁻¹⁻¹ = g⁻¹⁻¹ * 1 := by rw [mul_one]
    _ = g⁻¹⁻¹ * (g⁻¹ * g) := by rw [inv_mul_self]
    _ = (g⁻¹⁻¹ * g⁻¹) * g := by rw [mul_assoc]
    _ = 1 * g := by rw [inv_mul_self]
    _ = g := by rw [one_mul]
```

Note that in this case, the correct rewrite arguments are inferred through unification [Dowek 2001]. However, in `calc`, steps have to be given individually, which usually becomes tedious and error-prone. We would prefer to write a proof sketch like the one outlined in Section 1:

```
g⁻¹⁻¹ = g⁻¹⁻¹ * (g⁻¹ * g)
    _ = g
```

This section describes a prototype Lean 4 tactic[2] based on egg [Willsey et al. 2021], which enables guided equality saturation in Lean 4 and allows writing proofs similarly to the example above.

4.2 Implementing Guided Equality Saturation in Lean 4

To integrate this method into a theorem prover like Lean 4 [de Moura and Ullrich 2021], we use tactic metaprogramming to build the rewrites, the e-graph, saturate it and extract a proof out of it. This section outlines the main parts of the implementation. Lean 4 has a foreign-function interface that allows calling libraries like egg, but our prototype uses IO and operating system pipes instead.

The implementation first gathers the equalities passed to the tactic, as well as the equality we want to prove, `l = r`, and concretely its left-and right-hand sides, `l` and `r`. These are used to instantiate an e-graph using egg representing the two terms `l` and `r`, and the given rewrites. This implements the algorithm with terms as guides, as described in Section 3.2. As soon as the equivalence classes of the left-hand-side and right-hand-side get merged the procedure stops. In this case, we learn that `l` and `r` belong to the same equivalence class. We return the sequence of rewrites that witness `l = r`.

---

[2]https://github.com/opencompl/egg-tactic-code

In fact, rather than directly sending terms such as `l` and `r`, we send their internal encoding in Lean's `Expr` datatype. The `Expr` datatype represents terms in the Lean kernel and has constructs for constants, free variables, lambda abstractions, let-bindings, and function applications. These are serialized, sent to egg, and deserialized into the `Expr` datatype in Lean.

Rewrites are usually universally quantified. For example, associativity is expressed as `forall a b c : a * (b * c) = (a * b) * c`, i.e. universally quantified over the variables `a, b, c`. When looking through the passed arguments for terms with equalities, we first consider a term that begins with a universal quantifier. Since Lean uses a locally nameless [Charguéraud 2012] approach, we instantiate these quantifiers, which converts the quantified variables into free variables in an extended context. This exposes the bare equality that was behind universal quantification. We keep track of which free variables represent universally quantified variables that have now been instantiated due to Lean's locally nameless encoding. Then, when performing the encoding into egg, we communicate that these terms are, in fact, metavariables.

Finally, if egg finds a series of rewrites that relate both sides of the equation, we can use a reconstruction algorithm to obtain a witness of the equality from the e-graph [Flatt et al. 2022; Nieuwenhuis and Oliveras 2005]. Once we have constructed the series of rewrites with the corresponding instances of the applications, we can apply them to our goal within the Lean tactic. Crucially, Lean will then type check this and ensure the series of rewrites is sound. Hence, neither our tactic nor egg are part of the Trusted Computing Base (TCB), as a bug in either will only result in a failed tactic application and Lean will report that the goal was not proven. In practice, this also means that our tactic works as a proof checker for egg [Pnueli et al. 1998], by translation validation — our tactic translates the egg data structures into a proof that is verified by the Lean kernel.

4.2.1 Failing Gracefully with Simplification. If the equality saturation procedure fails to find an equivalence between the left and right-hand sides, all is not lost. For each side, we use the extraction mechanism to select a term with the smallest (AST) size in the corresponding equivalence class, along with the proof of equivalence between the smaller terms and their parents. This allows us to simplify the goal, potentially making progress towards a proof. In fact, it is not even necessary to have a goal of the form `l = r`. We can apply our technique to simplify a single term, much like the simplification tactics `simp` and `autorewrite` of Lean and Coq do, with all the additional rewrite capabilities from equality saturation. These simplification results may be, for example, used as guides in a guided equality saturation, to then continue the manual part of the search from there.

4.2.2 Limitations. The success of rewrite-based tactics like ours, or the greedy `simp`, hinges largely on being able to find rewrites that witness the equality of the left-hand and right-hand sides of the equation we are trying to prove. In fact, tactics like `simp` or Coq's `autorewrite` build entire tagging systems just to tag rewrites that might be relevant for them to use in simplification, instead of explicitly passing them. It is possible to go one step further and try to find a matching rewrite out of the theorems available; Sledgehammers in Isabelle do something similar, for example [Blanchette et al. 2011; Böhme and Nipkow 2010]. Our prototype implementation takes the rewrites as explicit arguments, and extending it with these kinds of systems is an orthogonal task.

Another limitation of our tactic is that it only supports the fragment of Lean 4 terms characterized by the following syntax:

```
e ::= named-constant | app e e | free-variable (type of terms we support)
eq ::= Eq e e | forall (x : α), eq
```

Here, `app` denotes function application and `Eq` denotes the equality type (with a single constructor `rfl a a` for all $a : \alpha$). This encoding is parametric over α, but only one type α can be instantiated at a time. The types of the terms, in particular, are also part of our encoding, as they are in Lean.

Table 1. Comparing different methods for proving the Knuth-Bendix Lemmas for groups.

tactic	inv_mul_c_left	mul_inv_c_left	one_inv	inv_mul	inv_inv
simp	✓	✓	✗	✗	✗
aesop	✓	✓	✗	✗	✗
unguided eqsat	✓	✓	✓	✗	✗
guided eqsat	✓	✓	✓	✓	✓
rw (manual)	✓	✓	✓	✓	✓
sledgehammer	✓	✓	✓	✓	✓

We currently do not add types to the metavariables to guard the rewrites, however. Lean's type checker will certainly not allow us to break things if the rewrite is applied to a term of the wrong type. However, there is nothing fundamentally preventing us from encoding our terms as typed expressions to avoid this issue in the future. Similarly, we do not rewrite over lambda expressions, nor support generalized rewriting. All of these are future work, but orthogonal to demonstrating guided equality saturation. There is in fact a work-in-progress port of the Lean 3 congruence closure tactic to Lean 4, independent of this work, that will address many of these issues[3].

4.3 Evaluating Guided Equality Saturation for Proving Theorems in Lean 4

To evaluate the tactic, we consider examples of increasing complexity.

4.3.1 Proving the Knuth-Bendix Lemmas for Groups. We first consider the example from group theory discussed in Section 1, as well as the rest of the additional lemmas from the Knuth-Bendix completion for groups. These are simple to evaluate, not requiring a large buildup of theory, yet useful as they provide a decision procedure for the word problem in (free) groups:

```
lemma inv_mul_c_left : a⁻¹ * (a * b) = b
lemma mul_inv_c_left : a * (a⁻¹ * b) = b
lemma one_inv : (1 : G)⁻¹ = 1
lemma inv_mul : (a * b)⁻¹ = b⁻¹ * a⁻¹
lemma inv_inv : a ⁻¹ ⁻¹ = a
```

Table 1 shows the comparison of different methods available in Lean to prove these lemmas with rewriting. We compare with `aesop`, the state of the art in proof automation in Lean [Limperg and From 2023] and `simp`. `simp` uses greedy rewriting; it does not follow an explicit cost function, it applies tagged rewrites marked as 'simp' until no further rewrite applies. `aesop` uses a tree-search to find tactic sequences that prove a goal. `aesop` relies on simp for the equational rewriting and does not expose rewrite choices and locations through its search tree. As a result, both tactics can prove only two of the four lemmas. Unguided equality saturation can apply rewrites in both directions and thus proves an additional one, `one_inv`. Both guided equality saturation and manual rewriting prove all lemmas, as they are provided with human insights. The main difference, however, is that in guided equality saturation only the key insights have to be provided, as motivated in Section 1. For example, we can prove `inv_inv` with a single guide, rather than 5 manual rewrite steps. We also compare to Isabelle's `sledgehammer` [Blanchette et al. 2013; Böhme and Nipkow 2010], which also closes all lemmas. This is not very surprising, since `sledgehammer` is much more powerful, combining multiple semi-decision procedures and including congruence closure as one of them.

---

[3]See https://github.com/leanprover-community/mathlib4/pull/5938

4.3.2 Larger Use Case: "Freshman's Dream". The previous examples are simple and require only a handful of rewrites each. To increase the complexity of the reasoning, without invoking esoteric mathematics, we consider examples from ring theory, where there are two operations (addition and multiplication) and thus significantly more rewrites available. For example, consider the theorem $(x + y)^2 = x^2 + y^2$, sometimes dubbed "freshman's dream", which holds in a commutative ring with characteristic 2. The proof using our tactic looks like this:

```
(x + y)^2 = (x + y) * (x + y) := by ges
        _ = x * (x + y) + y * (x + y) := by ges
        _ = x^2 + x * y + y * x + y^2 := by ges
        _ = x^2 + y^2 := by ges
```

which — if we ignore the somewhat verbose syntax, as we lack suitable syntactic sugar — corresponds roughly to what you would expect to see in a textbook proof. For comparison, if we use the same intermediate steps and try to prove the individual equalities with greedy rewriting, this does not yield a proof. On the other hand, a manual version that does all the rewriting explicitly requires 15 steps for this proof and is tediously verbose. Unguided equality saturation also finds a proof, but has a significantly longer runtime. Interestingly, while sledgehammer's calls to external solvers find proofs, neither metis [Hurd 2003; Paulson and Susanto 2007] nor Isabelle's smt tactics [Blanchette et al. 2013] are able to reconstruct a proof. The code for these two versions, as well as the ones for groups, can be found in the supplementary material.

Figure 9a compares the runtime, and number of steps required, for the manual version of rewrites, the guided equality saturation version above, and an unguided equality saturation version. We report the median runtimes of 10 executions, on an AMD Ryzen 9 3900X 12-Core machine with 132 GB of memory. The runtimes include the startup time of Lean, parsing, and the serialization and deserialization times when communicating with the egg-based library. As a baseline to measure the overhead, we also compare with a manual version with guided equality saturation using all 15 steps from the manual proof as guides. The manual version took $0.534 \pm 0.019s$, while this baseline version using all guides with eqsat took $0.540 \pm 0.015s$: barely slower, and within a standard deviation of each other. For reference, the textbook-guided version took $0.534 \pm 0.031s$, basically identical to the manual version. We see that the overhead even with this prototype implementation is negligible for these cases. Unguided equality saturation, on the other hand, takes about 4 minutes to prove this equality. This shows that while unguided equality saturation is powerful in principle, it is much less useful in practice, as it fails to prove simple theorems in a short period of time.
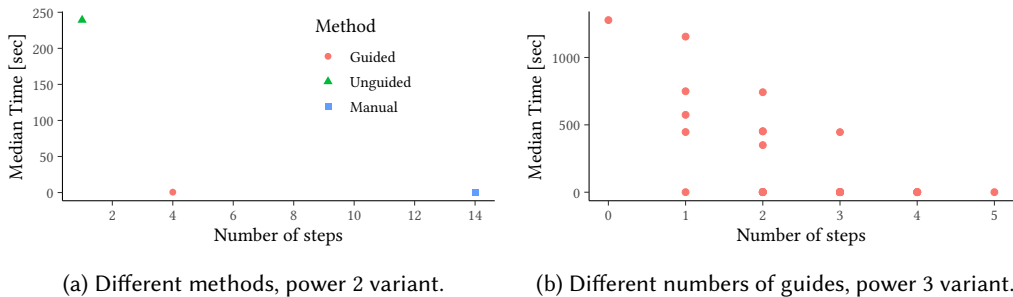

(a) Different methods, power 2 variant.

(b) Different numbers of guides, power 3 variant.

Fig. 9. Comparing runtimes and number of steps required for proving variants of the "freshman's dream" equality in Lean 4.

4.3.3 Proving $(x + y)^3 = x^3 + x * y^2 + x^2 * y + y$ for Rings. We can scale this up by considering the next power of the binomial, $(x + y)^3 = x^3 + x * y^2 + x^2 * y + y$, in characteristic 2. Adding two more guides, for multiplying out the remaining $(x + y)$, we still stay well below a second; in contrast, the unguided version takes over 20 minutes. For techniques like these to be useful in practice in a proof assistant, they need to be responsive and finish within seconds, at most.

To evaluate the effect of the guides, we take this proof sketch and remove the 5 intermediate guides, testing al 32 possible combinations of them. Figure 9b shows the results, where multiple points exist for each number of guides k, corresponding to the different ways of selecting k out of 5 guides. We see that, as expected, more guides seem to generally mean less time for finding a proof. Crucially, the right step can significantly speed everything up. This theorem can be proven in less than a second with a single well-chosen guide, but choosing a bad guide can mean almost no speedup at all. This corresponds to our intuition, some steps are easier to follow than others.

In summary, guided equality saturation allows us to write proof sketches that look like a textbook's, skipping steps and leaving out details, yet still reconstructing a full formal proof quickly.

4.3.4 Comparison to an Actual Textbook. We now look at an actual textbook with this result to compare, but we see it is proven as a special case of a more general theorem, using the binomial theorem [Rotman 2006]. Figure 10 shows the comparison between the Lean code and the textbook proof. We see that while the syntax is not as polished, it has essentially the same information as the textbook. Unfortunately, guided equality saturation does **not** complete this sketch if just given the ring axioms, like in the examples above. However, that is also not how we reconstruct the proof from our intuition either: consider the second step, which rewrites $(\frac{1}{(n-r+1)} + \frac{1}{r})$ to $(\frac{r+n-r+1}{r(n-r+1)})$. When reconstructing this step, we are implicitly using a lemma that tells us how to add fractions: $\frac{a}{b} + \frac{c}{d} = \frac{ad+bc}{db}$. This lemma does not hold for all $a, b, c, d \in R$ though, the textbook also uses the fact that $n - r + 1 \neq 0$, which follows non-trivially from $r \leq n$ in the integers, and the canonical ring homomorphism from them. If we give a handful of additional lemmas like this to our guided equality saturation tactic, with the satisfied preconditions, it can reconstruct the full equational proof from this textbook as well. That is to say, if we manually prove that $n - r + 1 > 0$ and then instantiate $1/(n - r + 1) + 1/r = r + n - r + 1/(r * (n - r + 1))$ explicitly for these concrete n and r, then we can reason about the equality using this manually-proven rewrite. However, our tactic cannot reason about these non-trivial preconditions yet. In future work we can explore the use of guarded rewrites (cf. [Willsey et al. 2021]) to achive this.

Fig. 10. Comparing a scan of the equational reasoning in an actual textbook proof in [Rotman 2006] and our Lean 4 version. We omit some boilerplate code in the Lean version for simplicity, as it lacks syntactic sugar. This part of the proof does not include reasoning about the the denominators being non-zero.

Guided equality saturation thus allows us to prove theorems with a semi-automatic approach that we could not prove with unguided equality saturation. For many cases where the full equality saturation does work, a few well-chosen guides can take a proof from several minutes down to under a second. Overall, this allows us to match the intuition often found in textbooks, where proof sketches skip steps and omit details, by reconstructing the full formal proof using these proof sketches as guides — at least when these work with purely equational reasoning.

5 CASE STUDY: PROGRAM OPTIMIZATION

Equality saturation has found many applications in program optimization [Smith et al. 2021; Tate et al. 2009; VanHattum et al. 2021; Wang et al. 2020; Yang et al. 2021]. Many applications tightly couple language and rewrite rule design with the equality saturation technique to mitigate e-graph growth, and often deliver impressive results.

This case study explores the limits of equality saturation for optimizing parallel linear algebra code in the RISE functional language, and how to overcome those limitations. First, we introduce RISE in Section 5.1. Then in Section 5.2 we describe how to encode the full language including variable bindings, which are often avoided for equality saturation. Finally, in Section 5.3 we optimize matrix multiplication as a case study performing seven complex program optimizations that required tens of thousands of manual rewrite steps in [Hagedorn et al. 2020]—beyond the scope of what prior applications of equality saturation have attempted.

5.1 Rewriting the RISE Functional Array Language

RISE [Hagedorn et al. 2020] is a functional array programming language. It is a spiritual successor of LIFT [Steuwer et al. 2015, 2017] that demonstrated performance portability across hardware by automatically applying semantics-preserving rewrite rules to optimize programs from various domains, including scientific code [Hagedorn et al. 2018] and convolutions [Mogers et al. 2020].

5.1.1 The RISE Language. RISE is well suited for rewrite-based optimizations as a functional, side-effect-free language. It is based on typed lambda calculus and thus provides lambda abstraction (λx. b), function application (f x), identifiers and literals. RISE expresses data-parallel computations as compositions of high-level computational patterns over dense multidimensional arrays (a.k.a. tensors). `map` applies a function to each element of an array. `reduce` combines all elements of an array to a single value given a binary reduction operator. `split`, `join`, `transpose`, `zip`, `unzip` and `slide` reshape arrays in various ways.

High-level programs, such as the matrix multiplication in the top left of Figure 11, specify computations without committing to a particular implementation. Implementation choices are explicitly encoded in RISE programs by applying rewrite rules that introduce low-level patterns that directly correspond to a particular implementation. For example, `reduceSeq` is a sequential reduction, and multiple low-level `map`-like patterns correspond to different sequential and parallel implementations. After the rewriting of a RISE program is complete, it is translated to low-level imperative code such as C or OpenCL for execution. The RISE program at the bottom of Figure 11 shows an optimized version of matrix multiplication. A common loop blocking optimization, that improves data locality and hence memory usage, has been introduced by rewriting.

5.1.2 Rewriting RISE Programs. RISE is complemented by a second language ELEVATE [Hagedorn et al. 2020] that allows programmers to manually describe complex optimizations as compositions of rewrite rules, called *rewriting strategies*. The performance of the code generated by RISE and ELEVATE is comparable with state-of-the-art compilers, e.g. with the TVM deep learning compiler [Chen et al. 2018] for matrix multiplication [Hagedorn et al. 2020]; and with the Halide image processing compiler [Ragan-Kelley et al. 2012] for the Harris corner detection [Koehler and Steuwer 2021].

```
1  map (λaRow.                                                     | for m:
2    map (λbCol.                                                   |   for n:
3      reduce + 0 (map (λy.(fst y) × (snd y)) (zip aRow bCol)))    |     for k:
4        (transpose b)) a
```

$$\longmapsto^*$$

```
1  join (map (map join) (map transpose
2    map                                                          | for m / 32:
3      (map λx2.                                                  |   for n / 32:
4        reduceSeq (λx3. λx4.                                     |     for k / 4:
5          reduceSeq λx5. λx6.                                    |       for 4:
6            map                                                  |         for 32:
7              (map (λx7.                                         |           for 32:
8                (fst x7) + (fst (snd x7)) × (snd (snd x7)))
9                (map (λx7. zip (fst x7) (snd x7)) (zip x5 x6)))
10               (transpose (map transpose (snd
11                 (unzip (map unzip map (λx5. zip (fst x5) (snd x5)) (zip x3 x4)))))))
12           (generate (λx3. generate (λx4. 0)))
13           transpose (map transpose x2))
14       (map (map (map (map (split 4)))) (map transpose (map
15         (map (λx2. map (map (zip x2) (split 32 (transpose b))))) split 32 a)))))
```

Fig. 11. A blocking optimization for matrix multiplication performed via rewriting in RISE. In the initial program (top), a dot product is computed between each row of a (aRow) and column of b (bCol). In the final program (bottom), a blocking optimization has been applied. Loops characteristic of the optimization are shown on the right of the | symbols, and are not part of the RISE program. The remaining program regions reshape input arrays, initialise arrays, compute with scalars, and reshape output arrays.

5.1.3 Limitations of Manual Rewriting with Strategies.

Although ELEVATE enables the development of abstractions that help with conciseness, strategies remain challenging to write. Fundamentally, ELEVATE delegates the problem of ordering thousands of rewrites to the human expert. Hagedorn et al. [2020] and Koehler and Steuwer [2021] estimate spending between two and five person-weeks developing the ELEVATE strategies for their matrix multiplication and image processing case studies.

This case study explores how to reduce the manual effort required to perform these optimizations by using guided equality saturation. No good heuristic to make local rewriting decisions is known for RISE, which is why we do not explore greedy rewriting and its variations [Kourta et al. 2022].

5.2 Rewriting RISE with Equality Saturation

Rewriting RISE with equality saturation requires encoding RISE terms and rewrite rules in an equality saturation framework like egg. This poses challenges as RISE is based on the lambda calculus, and its rewrite rules involve name bindings, substitutions, and freshness predicates.

Figure 12 shows the familiar *map* fusion and fission rules, as well as the standard β- and η-reduction rules of lambda calculus. In contrast to the rules of Section 2.1, name bindings are used instead of function composition. Dealing with name bindings in equality saturation is an open challenge [Willsey et al. 2021]. Lambda calculus programs with name bindings can be encoded as combinatory logic terms, but the translation results in a term of size $O(n^3)$ in the worst case [Lachowski 2018]. The language and rewrite rules could be redesigned to avoid name bindings, as in Smith et al. [2021], but here we wish to minimize changes to RISE and the rewrite rules.

Here we adopt the techniques explored by Koehler [2022] to efficiently encode a polymorphically typed lambda calculus. In practice, these techniques reduce the runtime and memory consumption of equality saturation by orders of magnitude when optimizing RISE programs. We now give a quick overview of these techniques.

$$map \; f \; (map \; g \; arg) \longmapsto map \; (\lambda x. \; f \; (g \; x)) \; arg \qquad\qquad\qquad \text{(map-fusion)}$$
$$map \; (\lambda x. \; f \; gx) \longmapsto \lambda y. \; map \; f \; (map \; (\lambda x. \; gx) \; y) \quad \text{if } x \text{ not free in } f \qquad \text{(map-fission)}$$
$$(\lambda x. \; b) \; e \longmapsto b[e/x] \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(β-reduction)}$$
$$\lambda x. \; f \; x \longmapsto f \qquad\qquad\qquad\qquad\qquad \text{if } x \text{ not free in } f \qquad \text{(η-reduction)}$$

Fig. 12. RISE rewrite rules using substitution, name bindings (lambda abstractions), and freshness predicates.

5.2.1 Dealing with Substitution. The β-reduction rule requires substituting $b[e/x]$. Standard term substitution cannot be used directly during equality saturation, as the b and e pattern variables are not matched by terms, but by e-classes.

A simple way to address this is to use *explicit substitution* as in egg's lambda calculus example [Willsey et al. 2021]: a syntactic constructor is added representing substitutions, along with rewrite rules to encode its small-step behaviour. Unfortunately, explicit substitution adds all intermediate steps to the e-graph, quickly exploding its size.

To avoid this effect, *extraction-based substitution* is used, an approximation that works as follows:

(1) extract a term for each e-class involved in the substitution (i.e. b and e);
(2) perform standard term substitution;
(3) add the resulting term to the e-graph.

5.2.2 Dealing with Name Bindings. During equality saturation, inappropriate handling of name bindings leads to serious efficiency issues. Consider rules like map fusion that create a new lambda abstraction on their right-hand side. What name should be introduced when they are applied?

Generating a fresh name using a global counter (aka. gensym) is a common solution in standard term rewriting [Augustsson et al. 1994]. However, such an approach quickly burdens the e-graph with many α-equivalent terms.[4]

Instead, De Bruijn indices [de Bruijn 1972] are used to represent lambda terms without naming the bound variables. The more user-friendly name-based rewrite rules are automatically translated to the index-based rules used internally [Bonelli et al. 2000]. Translations to use indices are computationally inexpensive: they are linear in term size, and performed outside of the guided search hot loop. It is only necessary to convert the starting term and the sets of rewrite rules before starting the guided search, and to convert the final term once the search completes.

5.2.3 Dealing with Freshness Predicates. Handling predicates is also non-trivial in equality saturation. The η-reduction rule has the side condition "if x not free in f", but in an e-graph f is an e-class and not a term. Following egg's lambda calculus example [Willsey et al. 2021], we only apply the rule if $\forall t \in f. \; x$ not free in t. This method is efficient but is an approximation. In our case study, we have neither observed substitution nor freshness predicate approximations to be an issue.

5.2.4 Further Considerations. Types are also important for RISE rewrite rules, we, therefore, embed types in the e-graph, associating a type with each e-class, that all of its e-nodes must satisfy. The more user-friendly partially-typed RISE rewrite rules are automatically translated to the explicitly typed rules used internally. Types are inferred by first inferring the types on the left-hand side, before checking that the right-hand side is well-typed for any well-typed left-hand side.

---

[4]Two λ terms are α-equivalent if one can be made equivalent to the other simply by renaming variables.

Finally, certain program properties are required in RISE to obtain a valid low-level program from which imperative code can be generated. We fully automate enforcing such properties in a final cleanup phase. For example, sequential loops and memory copies are inserted where required, and let expressions are hoisted as much as possible.

5.3 Evaluating Guided Equality Saturation for Optimizing Programs in RISE

This section compares guided and unguided equality saturation performing complex RISE program optimizations. We evaluate seven typical compiler optimizations, including loop blocking, loop permutation, vectorization, and multithreading, described in the TVM manual.[5] They have been reproduced by Hagedorn et al. [2020] using manually specified ELEVATE strategies that express the optimizations as compositions of rewrites achieving the same high performance as TVM.

We compare the runtime and memory requirements for unguided and for guided equality saturation. In both cases the final optimization goal is specified as a sketch, allowing for slightly different programs to be found. We validate the performance of the optimized code by checking that the generated C code is equivalent, modulo variable names, to the code obtained using the manual ELEVATE strategies. The generated C code is provided in the paper's supplementary material.

5.3.1 Experimental Setup. We have implemented an equality saturation engine inspired by egg in Scala[6], allowing close integration with the existing RISE codebase, instead of interfacing directly with egg via Rust-Scala interoperability. The standard Java utilities are used for measurements: System.nanoTime() to measure search runtime, and the Runtime API to approximate maximum heap memory residency with regular sampling.

Platforms. The experiments are performed on two platforms. For manual ELEVATE strategies and our sketch-guided equality saturation, we use a less powerful AMD Ryzen 5 PRO 2500U with 4 GB of RAM available to the JVM. For unguided equality saturation, we use a more powerful Intel Xeon E5-2640 v2 with 60 GB of RAM available to the JVM.

Rewrite Rule Scheduling. By default, the egg library uses a BackoffScheduler preventing specific rules from being applied too often and reducing e-graph growth in the presence of "explosive" rules such as associativity and commutativity. Our experience with RISE optimization is that using the BackoffScheduler is counterproductive as the desired optimization depends on some explosive rules. For this reason, and to make result analysis easier, we do not use a rewrite rule scheduler.

5.3.2 Matrix Multiplication Optimizations. We investigate seven increasingly complicated matrix multiplication optimization goals. Each goal incrementally adds more optimizations.
- The *baseline* goal uses 3 straightforward nested loops to perform the matrix multiplication.
- The *blocking* goal adds a blocking (or tiling) optimization for improved data locality, resulting in 6 nested loops where the 3 innermost ones process $4 \times 32 \times 32$ blocks.
- The *vectorization* goal adds parallelism by vectorizing the innermost loop over 32 elements.
- The *loop-perm* goal changes the order of the 6 nested loops, for improved data locality.
- The *array-packing* goal adds intermediate storage for the transposed b matrix, improving memory access patterns.
- The *cache-blocks* goal unrolls the inner reduction loop.
- The *parallel* goal adds parallelism by multi-threading the outermost loop.

---

[5]https://tvm.apache.org/docs/how_to/optimize_operators/opt_gemm.html
[6]https://github.com/rise-lang/shine/blob/sges/src/main/scala/rise/eqsat

5.3.3   Runtime and Memory Consumption of (Un)Guided Equality Saturation.

Unguided Equality Saturation. Table 2 shows the runtime and memory consumption required to find the optimization goals with unguided equality saturation. The search terminates when the sketch describing the optimization goal is found in the e-graph.

The 5 most complex optimization goals are not found before exhausting the 60 GB of available memory. Only the *baseline* and *blocking* goals are found, and the search for *blocking* requires more than 1 h and about 35 GB of RAM. Millions of rewrite rules are applied, and the e-graph contains millions of e-nodes and e-classes. More complex optimizations involve more rewrite rules, creating a richer space of equivalent programs but exhausting memory faster. As examples, *vectorization* and *loop-perm* use vectorization rules, while *array-packing*, *cache-blocks*, and *parallel* use rules for optimizing memory storage.

Sketch-Guided Equality Saturation. Table 3 shows the runtime and memory consumption for sketch-guided equality saturation with 1, 2 or 3 sketch guides.

All optimizations are found in less than 10 s using less than 0.5 GB of RAM. Interestingly the number of rewrite rules applied by sketch-guided equality saturation is in the same order of magnitude as for the manual Elevate strategies reported in [Hagedorn et al. 2020]. On one hand, equality saturation applies more rules than necessary because of its explorative nature. On the other hand, Elevate strategies apply more rules than necessary because they re-apply the same rule to the same sub-expression and do not necessarily orchestrate the shortest possible rewrite path. The e-graphs contain of the order of 10^4 e-nodes and e-classes, two orders of magnitude less than the 10^6 required for *blocking* without sketch-guidance.

Table 2. Runtime and memory consumption for **unguided equality saturation** with efficient lambda calculus encoding. Only the *baseline* and *blocking* optimization goals are found, with other optimizations exceeding 60 GB.

goal	found?	runtime	RAM	rules	e-nodes	e-classes
baseline	✓	0.5 s	0.02 GB	2	51	49
blocking	✓	>1 h	35 GB	5 M	4 M	2 M
vectorization	✗	>1 h	>60 GB			
loop-perm	✗	>1 h	>60 GB			
array-packing	✗	35 m	>60 GB			
cache-blocks	✗	35 m	>60 GB			
parallel	✗	35 m	>60 GB			

Table 3. Runtime and memory consumption for **sketch-guided equality saturation** with efficient lambda calculus encoding. All optimizations are found in seconds using less than 0.5 GB of memory and requiring at most 3 sketch guides.

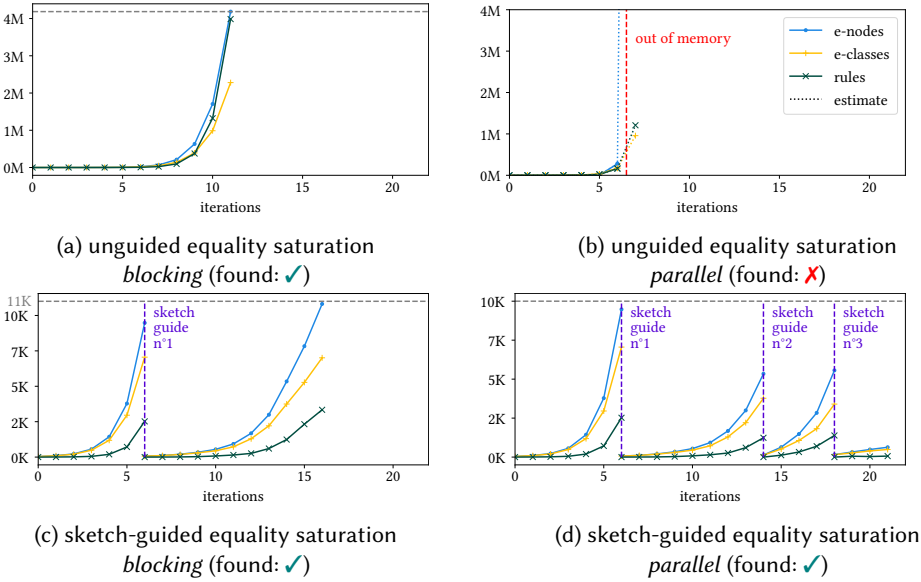| goal	sketch guides	found?	runtime	RAM	rules	e-nodes	e-classes
baseline	0	✓	0.5 s	0.02 GB	2	51	49
blocking	1	✓	7 s	0.3 GB	11 K	11 K	7 K
vectorization	2	✓	7 s	0.4 GB	11 K	11 K	7 K
loop-perm	2	✓	4 s	0.3 GB	6 K	10 K	7 K
array-packing	3	✓	5 s	0.4 GB	9 K	10 K	7 K
cache-blocks	3	✓	5 s	0.5 GB	9 K	10 K	7 K
parallel	3	✓	5 s	0.4 GB	9 K	10 K	7 K

Fig. 13. The evolution of the e-graph, and the number of rewrite rules applied, during searches for two optimization goals. Sketch guides are depicted with purple vertical lines. Note that the scale of the y-axes for unguided graphs (a) and (b) is millions, while for guided graphs (c) and (d) it is thousands.

5.3.4 E-Graph Evolution in (Un)Guided Equality Saturation. Figure 13 plots the growth of the e-graphs during unguided and sketch-guided searches for the *blocking* and *parallel* optimization goals from Tables 2 and 3. The e-graphs produced by unguided equality saturation grow exponentially with each search iteration. The e-graph contains millions of e-nodes and e-classes after applying millions of rules within a few iterations: less than 10. Such rapid growth limits the scalability of unguided search, for example in the 7th iteration of the *parallel* search, the e-graph exhausts 60 GB of memory.

While the e-graphs produced with sketch guidance typically also grow exponentially with each iteration, each sketch is satisfied within a few iterations thanks to an appropriate selection of sketches. The number of rewrites and the maximum e-graph size is *three orders of magnitudes smaller* than for unguided search: no more than 11 K in our example searches. Once a program satisfying a sketch guide is found, a new search is started for the next sketch using that program, growing a fresh e-graph. Sketch-guidance enables scaling to more complex optimizations, such as *parallel*, by factoring optimizations into as many sketch-guided searches as necessary.

The search for the final *parallel* sketch goal shows linear rather than exponential growth, as the rewrite rules selected for the search have little interaction.

5.3.5 Human Guidance Provided.

Sketch-Guides. Table 4 shows how each optimization goal is achieved in logical steps, each corresponding to a sketch describing the program after the step is applied. It transpires that a *split* sketch similar to the one shown in Section 3.3 is a useful first guide for all goals. While the sketch sizes range from 7 to 12 operators, programs are of size 90 to 124 operators, showing that each sketch elides around 90% of the RISE program. Even when 4 sketches must be written, the total sketch size is still small: the largest total being 38 operators.

Table 4. Decomposition of each optimization goal into logical steps. A sketch is defined for each logical step. In this table, sketch size counts operators such as `containsMap`, program size counts operators such as `map`, lambdas, variables and constants: not λ applications.

goal	sketch guides	sketch goal	sketch size	program size
blocking	*split*	*reorder$_1$*	7	90
vectorization	*split + reorder$_1$*	*lower$_1$*	7	124
loop-perm	*split + reorder$_2$*	*lower$_2$*	7	104
array-packing	*split + reorder$_2$ + store*	*lower$_3$*	7-12	121
cache-blocks	*split + reorder$_2$ + store*	*lower$_4$*	7-12	121
parallel	*split + reorder$_2$ + store*	*lower$_5$*	7-12	121

The paper's supplementary material contains all handwritten sketches as well as examples of discovered RISE programs. Some intricate program aspects never need to be specified in the sketches, for example, array reshaping patterns such as `split`, `join` and `transpose`.

Choice of Rules and Cost Model. Besides the sketches, programmers also specify the rules used in each search and a cost model. For the *split* sketch, 8 rules explain how to split `map` and `reduce`. The *reorder* sketches require 9 rules that swap various nestings of `map` and `reduce`. The *store* sketch requires 4 rules and the *lower* sketches 10 rules, 6 rules for vectorization, 1 rule for loop unrolling and 1 rule for loop parallelization. If we naively use all rules for the blocking search, the search runtime increases by about 25×, still finding the goal in minutes but showing the importance of selecting a small set of rules.

A simple cost model that minimizes weighted term size is used. For example, we increase `mapPar` weight to avoid implicit multi-threading. This cost model is only effective when combined with the guides requesting loop splitting, temporary storage, and parallelization. On its own, this cost model is a poor greedy rewriting heuristic, as the optimizations increase weighted term size. Rules and cost models may be reused and packaged into libraries for recurring logical steps.

6 RELATED WORK

Equality Saturation. Equality saturation [Tate et al. 2009; Willsey et al. 2021] has been proposed as an optimization technique based on e-graphs. E-graphs were originally designed for efficient congruence closure [de Moura and Bjørner 2008; Nelson 1980]. Besides theorem proving and program optimization, e-graphs are also useful in other settings, such as program synthesis and semantic code search [Premtoon et al. 2020]. In this paper, we propose guided equality saturation as a semi-automatic technique that mitigates scaling problems of equality saturation. Alternative approaches have been used to scale equality saturation in practice, as discussed in Section 2.3.2.

Sketches. While we propose using incomplete program sketches to guide rewriting, sketching has also been used for synthesizing programs. In [Lezama 2008], sketches are used with counterexample-guided inductive synthesis that combines a synthesizer with a validation procedure. Our approach differs as we use sketches for program rewriting rather than program synthesis. We use sketches as program patterns to filter a set of equivalent programs generated via equality saturation, and as a result do not require a validation procedure.

An automatic optimization procedure for TVM [Zheng et al. 2020] generates sketches before sampling low-level details, but does not use sketches as a vector for human insight.

Also related are Wiedijk [2003]'s formal proof sketches, which are in-line with our motivation for the theorem proving use-case of guided equality saturation. The scope and vision of these formal proof sketches is much larger, of course, as guided equality saturation is limited to equational reasoning. Recently, Jiang et al. [2023]'s "draft, sketch, prove" also used a related notion of proof

sketches aided with machine learning to close the gap between full formal proofs and informal ones, using formal proof sketches as an intermediate. Our guides can be seen in this context as the sketches, and guided equality saturation is an efficient technique to prove sketches of this kind.

Interactive Theorem Proving. Equational reasoning is important in Interactive Theorem Proving. Tactic languages [Gordon et al. 1979] allow experts to specify proofs manually. Most interactive theorem provers have a form of automatic greedy rewriting and usually also congruence closure [Corbineau 2006; Gjørup and Spitters 2020; Nieuwenhuis and Oliveras 2005; Selsam and de Moura 2016]. The idea of verifying an external solver is also established in provers, with integrations like SMTCoq [Ekici et al. 2017] or Isabelle's Sledgehammer [Blanchette et al. 2013]. Isabelle's Sledgehammer [Böhme and Nipkow 2010] integrates multiple automatic theorem provers, like SPASS [Weidenbach et al. 2009], Vampire [Riazanov and Voronkov 1999] or Zipperposition [Bentkamp et al. 2023]. Sledgehammer also has heuristics for finding appropriate lemmas, and even counter-examples when writing a wrong statement [Blanchette et al. 2011]. Such heuristics would also be very useful in the context of our Lean tactic, as outlined in Section 4. Indeed, our design is heavily influenced by Sledgehammer and SMTCoq. A concrete difference is that we don't reconstruct the proofs at the source-level [Paulson and Susanto 2007] like Sledgehammer does, which effectively means the external solvers just work to prune the space of lemmas required to find a proof [Böhme and Nipkow 2010]. Instead, we reconstruct the proof from the series of rewrites that witness the equivalence [Flatt et al. 2022; Nieuwenhuis and Oliveras 2005].

To the best of our knowledge, none of these tools integrate equality saturation, the specific focus here. However, many of the tools share principles and even data structures with equality saturation and are also useful for equational reasoning. With these tools, it may be feasible to provide guidance for congruence closure, superposition, SMT, or other operations with a view to obtaining similar benefits. However, the technical challenges and trade-offs are likely to be different. For example, it is not clear how to simplify terms (c.f. Section 4.2.1), or how to use sketch guides without the term extraction from equality saturation.

There is a sense in which our approach can be compared more to structured proof languages, like the calculational proof style in Isar [Nipkow 2002] or equational reasoning in Agda (cf. [Kokke et al. 2020]). Indeed, by writing proof steps with the sketches in Isar's "calculational style" and calling sledgehammer on each subgoal, we can manually reproduce something very similar to the Lean tactic described in this paper. This is also enough to prove the examples from ring theory in Section 4 where using sledgehammer alone did not work. While there are syntactic differences between the two, these techniques are very similar. The different backends of sledghammer are probably more powerful in general, and will prove a larger set of theorems automatically. Guided equality saturation, on the other hand, allows you to optimize terms (our tactic does this for simplification). In future work we may use sketch-guides instead of terms for more flexible proof sketches, and guarded rewrites to deal with preconditions. These are unique to sketch-guided equality saturation.

Program Optimization. Historically, programmers had either to explicitly write optimized code, e.g., explicit vectorization or loop ordering, or to entrust optimization to a black box compiler.

Black box compilers use fully automatic optimization techniques such as greedy rewriting [Jones et al. 2001; Lattner and Adve 2004], equality saturation [Tate et al. 2009; Yang et al. 2021] or heuristic searches [Mullapudi et al. 2015; Steuwer et al. 2015]. Although full automation can sometimes yield high performance, it is not always feasible or even desirable, as it may result in poor performance or may be too time-consuming [Maleki et al. 2011; Parello et al. 2004]. In particular, greedy rewriting is used in compilers like GHC and LLVM because it is fast. However, ordering greedy rewriting phases to maximize performance is a difficult problem, known as the phase ordering problem [Touati and Barthou 2006]. Moreover, creating effective optimization heuristics is a challenging task that the community seeks to automate [Cummins et al. 2017; Stephenson et al. 2003].

When automatic optimization is unsatisfactory, programmers often fall back to manual optimization in order to achieve their performance goals [Lemaitre et al. 2017; Niittylahti et al. 2002].

More recently, programmers may control optimization through transformation scripts [Chen et al. 2008; Cohen et al. 2005], rewriting strategies [Hagedorn et al. 2020; Koehler and Steuwer 2021; Visser et al. 1998] or scheduling APIs [Chen et al. 2018; Ragan-Kelley et al. 2012]. As discussed in Section 5.1.3, precisely controlling optimization is challenging, quickly requiring excessive programmer effort. Guided optimization is a middle ground that aims to combine the productivity of automatic optimization with the flexibility of controlled optimization. Programmers may guide optimization within tools for the polyhedral framework [Zinenko 2016] or for scheduling APIs [Ikarashi et al. 2021]. To the best of our knowledge, this paper proposes the first guided optimization technique that uses program sketches as guides.

7 DISCUSSION: USER EXPERIENCE

Creating appropriate (sketch) guides is the key creative step in using guided equality saturation: be it for optimization, proof, or something else. This is akin to explaining program optimizations using incomplete program snippets; or explaining proofs using incomplete proof sketches. To understand such explanations, human readers must have the implicit background that enables them to fill in the gaps. With guided equality saturation, this background is explicitly encoded as sets of rewrite rules and cost functions.

Proposed Methodology. Guided equality saturation users can explore using different guides, rewrite rules and cost functions with a view to achieving their rewrite goal. They may well start with their rewrite goal and include all potentially useful rewrites. At first the search may fail by:

(1) Exhausting resources, either time or memory: there is insufficient guidance to make the search feasible, or there is no solution.
(2) Saturating without finding a solution: the goal is unreachable with the given set of rules.
(3) Finding an unexpected solution: the goal is too vague or the rules too permissive.

When a search fails, the user can iteratively apply some of the following steps:

(1) Adding more guides. This splits the search into simpler, more achievable, searches.
(2) Removing misguides. An incorrect guide may obstruct the search. One way to detect misguides is to try finding all guides at once: if a later guide is found more easily, then the earlier guide is likely a misguide.
(3) Adding rewrite rules. Forgetting to explicitly encode background commonly leads to failures.
(4) Removing unnecessary rewrite rules. This both speeds up the search and reduces the size of the e-graph (Section 5.3.5). Moreover it allows the use of less precise sketch guides and cost functions.
(5) Changing sketch precision. An overly precise sketch excludes valid terms with a slightly different structure. An overly vague sketch includes undesirable terms (Section 3.3.2).
(6) Changing the cost function. If a search succeeds but extracts a different term than expected, using a better cost function may solve the problem.

Improving the User Experience. To be useful, guided equality saturation must require less human effort than manual term rewriting, e.g. with rewriting strategies or tactics. While our intuition and case studies suggests that this is true in many scenarios, we have not attempted to quantify and contrast the human effort required. User studies could provide evidence of reduced effort, and insights into how to further improve user experience. Future work may investigate how to build interactive tools for guided equality saturation, and how to provide effective feedback: e.g.

visualizing the cost and result of the searches, or suggesting one of the iterative steps previously listed using machine learning.

Additional features might also improve user experience. For example, manipulating a tree of guides instead of a sequence may encourage more experimentation. SKETCHBASIC includes the constructs that we found useful for our case studies. Different applications however, may benefit from other constructs like named holes or sketch intersection. This paper does not present a universal sketch language that would serve a wide range of applications, as expressivity and extraction cost must be considered for a given application. As with regular expressions or database queries, more powerful constructs tend to require more expensive algorithms: this is a problem worth studying on its own.

In some domains developing a library of guides, sets of rewrite rules, and cost functions could also improve user experience. The supplementary material illustrates how we have defined sketch abstractions and reused them across all of the guides in Table 4. Moreover we find that some guides are useful across different goals, and discuss how four sets of rewrite rules are reused across searches. Future work could extend this to investigate the design of generic guides, sets of rewrite rules, and cost functions for reuse across multiple searches.

8 CONCLUSION

This paper explores an effective trade-off between the painstaking manual control of rewriting and automated, but often unsuccessful, rewriting. The motivating intuition is that humans often explain rewriting using (potentially incomplete) terms that identify the key rewrite stages, omitting a large number of intermediate routine rewrites. Specifically, we propose guided equality saturation, a semi-automatic rewriting technique, to scale beyond fully-automatic rewriting with limited human effort. For problems requiring long rewrite sequences, or some specific insight, such that fully automated equality saturation fails, a human provides guides. The guides are intermediate rewrite goals and may be either complete terms or incomplete sketches. With guidance, rewriting scales as it is decomposed into a sequence of equality saturations, each of feasible cost (Section 3).

We demonstrate the generality and effectiveness of guided equality saturation using case studies in theorem proving and program optimization. Using guided equality saturation as a novel tactic in the Lean 4 proof assistant allows proofs to be written in the style of textbook proof sketches, i.e., as a series of calculations that omit details and skip steps. With the use of guides, we find more proofs than unguided equality saturation. When both find a proof the guides reduce search time from minutes to fractions of a second (Section 4). Using guided equality saturation to optimize programs in the RISE functional array language enables optimizations that cannot be achieved with unguided equality saturation within an hour and using 60 GB of memory. Using no more than 3 guides, the same optimizations are achieved within seconds and using less than 1 GB of memory (Section 5).

We believe that guidance will increase the scale of the problems that can be solved with equality saturation across application domains. For theorem proving, we hope that equality saturation can bring scalable sketch-guided proof-search into day-to-day proof development. For program optimization, we envision performance engineers guiding compilers to generate better code using program sketches. In both domains, we expect problems to be resolved more effectively due to a powerful interplay between human intuition and scalable rewriting.

REFERENCES

Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to optimize halide with tree search and random programs. *ACM Trans. Graph.* 38, 4 (2019), 121:1–121:12. https://doi.org/10.1145/3306346.3322967

Luke Anderson, Andrew Adams, Karima Ma, Tzu-Mao Li, Tian Jin, and Jonathan Ragan-Kelley. 2021. Efficient automatic scheduling of imaging and vision pipelines for the GPU. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–28. https://doi.org/10.1145/3485486

Lennart Augustsson, Mikael Rittri, and Dan Synek. 1994. On Generating unique Names. *J. Funct. Program.* 4, 1 (1994), 117–123. https://doi.org/10.1017/S0956796800000988

Franz Baader and Tobias Nipkow. 1998. *Term rewriting and all that.* Cambridge University Press.

Leo Bachmair and Harald Ganzinger. 1994. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *J. Log. Comput.* 4, 3 (1994), 217–247. https://doi.org/10.1093/LOGCOM/4.3.217

Alexander Bentkamp, Jasmin Blanchette, Visa Nummelin, Sophie Tourret, Petar Vukmirovic, and Uwe Waldmann. 2023. Mechanical Mathematicians. *Commun. ACM* 66, 4 (2023), 80–90. https://doi.org/10.1145/3557998

Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. 2013. Extending Sledgehammer with SMT Solvers. *J. Autom. Reason.* 51, 1 (2013), 109–128. https://doi.org/10.1007/S10817-013-9278-5

Jasmin Christian Blanchette, Lukas Bulwahn, and Tobias Nipkow. 2011. Automatic Proof and Disproof in Isabelle/HOL. In *Frontiers of Combining Systems, 8th International Symposium, FroCoS 2011, Saarbrücken, Germany, October 5-7, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6989)*, Cesare Tinelli and Viorica Sofronie-Stokkermans (Eds.). Springer, 12–27. https://doi.org/10.1007/978-3-642-24364-6_2

Sascha Böhme and Tobias Nipkow. 2010. Sledgehammer: Judgement Day. In *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6173)*, Jürgen Giesl and Reiner Hähnle (Eds.). Springer, 107–121. https://doi.org/10.1007/978-3-642-14203-1_9

Eduardo Bonelli, Delia Kesner, and Alejandro Ríos. 2000. A de Bruijn Notation for Higher-Order Rewriting. In *Rewriting Techniques and Applications, 11th International Conference, RTA 2000, Norwich, UK, July 10-12, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1833)*, Leo Bachmair (Ed.). Springer, 62–79. https://doi.org/10.1007/10721975_5

Kevin Buzzard, Johan Commelin, and Patrick Massot. 2020. Formalising perfectoid spaces. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, Jasmin Blanchette and Catalin Hritcu (Eds.). ACM, 299–312. https://doi.org/10.1145/3372885.3373830

Arthur Charguéraud. 2012. The Locally Nameless Representation. *J. Autom. Reason.* 49, 3 (2012), 363–408. https://doi.org/10.1007/S10817-011-9225-2

Chun Chen, Jacqueline Chame, and Mary Hall. 2008. *CHiLL: A framework for composing high-level loop transformations.* Technical Report. Citeseer.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

Albert Cohen, Marc Sigler, Sylvain Girbal, Olivier Temam, David Parello, and Nicolas Vasilache. 2005. Facilitating the search for compositions of program transformations. In *Proceedings of the 19th Annual International Conference on Supercomputing, ICS 2005, Cambridge, Massachusetts, USA, June 20-22, 2005*, Arvind and Larry Rudolph (Eds.). ACM, 151–160. https://doi.org/10.1145/1088149.1088169

Pierre Corbineau. 2006. Deciding Equality in the Constructor Theory. In *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 4502)*, Thorsten Altenkirch and Conor McBride (Eds.). Springer, 78–92. https://doi.org/10.1007/978-3-540-74464-1_6

Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. End-to-End Deep Learning of Optimization Heuristics. In *26th International Conference on Parallel Architectures and Compilation Techniques, PACT 2017, Portland, OR, USA, September 9-13, 2017*. IEEE Computer Society, 219–232. https://doi.org/10.1109/PACT.2017.24

N.G de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (1972), 381–392.

Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12699)*, André Platzer and Geoff Sutcliffe (Eds.). Springer, 625–635. https://doi.org/10.1007/978-3-030-79876-5_37

Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2007. Efficient E-Matching for SMT Solvers. In *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4603)*, Frank Pfenning (Ed.). Springer, 183–198. https://doi.org/10.1007/978-3-540-73595-

3_13

Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Nachum Dershowitz. 1993. A Taste of Rewrite Systems. In *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991-1992, McMaster University, Hamilton, Ontario, Canada (Lecture Notes in Computer Science, Vol. 693)*, Peter E. Lauer (Ed.). Springer, 199–228. https://doi.org/10.1007/3-540-56883-2_11

Gilles Dowek. 2001. Higher-Order Unification and Matching. In *Handbook of Automated Reasoning (in 2 volumes)*, John Alan Robinson and Andrei Voronkov (Eds.). Elsevier and MIT Press, 1009–1062. https://doi.org/10.1016/B978-044450813-3/50018-7

Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W. Barrett. 2017. SMTCoq: A Plug-In for Integrating SMT Solvers into Coq. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10427)*, Rupak Majumdar and Viktor Kuncak (Eds.). Springer, 126–133. https://doi.org/10.1007/978-3-319-63390-9_7

Trevor Evans. 1978. Word problems. *Bull. Amer. Math. Soc.* 84, 5 (1978), 789–802.

Oliver Flatt, Samuel Coward, Max Willsey, Zachary Tatlock, and Pavel Panchekha. 2022. Small Proofs from Congruence Closure. In *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, Alberto Griggio and Neha Rungta (Eds.). IEEE, 75–83. https://doi.org/10.34727/2022/ISBN.978-3-85448-053-2_13

Emil Holm Gjørup and Bas Spitters. 2020. Congruence closure in cubical type theory. In *Workshop on Homotopy Type Theory/Univalent Foundations. https://www. cs. au. dk/˜ spitters/Emil. pdf*.

Georges Gonthier et al. 2008. Formal proof–the four-color theorem. *Notices of the AMS* 55, 11 (2008), 1382–1393.

Michael J Gordon, Arthur J Milner, and Christopher P Wadsworth. 1979. *Edinburgh LCF: a mechanised logic of computation*. Springer.

Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. 2020. Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies. *Proc. ACM Program. Lang.* 4, ICFP (2020), 92:1–92:29. https://doi.org/10.1145/3408974

Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High performance stencil code generation with lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, Vösendorf / Vienna, Austria, February 24-28, 2018*, Jens Knoop, Markus Schordan, Teresa Johnson, and Michael F. P. O'Boyle (Eds.). ACM, 100–112. https://doi.org/10.1145/3168824

Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Hoang Le Truong, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen, et al. 2017. A formal proof of the Kepler conjecture. In *Forum of mathematics, Pi*, Vol. 5. Cambridge University Press.

Jieh Hsiang, Hélène Kirchner, Pierre Lescanne, and Michaël Rusinowitch. 1992. The Term Rewriting Approach to Automated Theorem Proving. *J. Log. Program.* 14, 1&2 (1992), 71–99. https://doi.org/10.1016/0743-1066(92)90047-7

Joe Hurd. 2003. First-order proof tactics in higher-order logic theorem provers. *Design and Application of Strategies/Tactics in Higher Order Logics, number NASA/CP-2003-212448 in NASA Technical Reports* (2003), 56–68.

Yuka Ikarashi, Jonathan Ragan-Kelley, Tsukasa Fukusato, Jun Kato, and Takeo Igarashi. 2021. Guided Optimization for Image Processing Pipelines. In *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2021, St Louis, MO, USA, October 10-13, 2021*, Kyle J. Harms, Jácome Cunha, Steve Oney, and Caitlin Kelleher (Eds.). IEEE, 1–5. https://doi.org/10.1109/VL/HCC51201.2021.9576341

Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timothée Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. 2023. Draft, Sketch, and Prove: Guiding Formal Theorem Provers with Informal Proofs. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/pdf?id=SMa9EAovKMC

Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. 2001. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell workshop*, Vol. 1. 203–233.

Donald E. Knuth. 1977. A Generalization of Dijkstra's Algorithm. *Inf. Process. Lett.* 6, 1 (1977), 1–5. https://doi.org/10.1016/0020-0190(77)90002-3

D. E. Knuth and P. B. Bendix. 1983. *Simple Word Problems in Universal Algebras*. Springer Berlin Heidelberg, Berlin, Heidelberg, 342–376. https://doi.org/10.1007/978-3-642-81955-1_23

Thomas Koehler. 2022. *A domain-extensible compiler with controllable automation of optimisations*. Ph. D. Dissertation. University of Glasgow, UK. https://doi.org/10.5525/GLA.THESIS.83323

Thomas Koehler and Michel Steuwer. 2021. Towards a Domain-Extensible Compiler: Optimizing an Image Processing Pipeline on Mobile CPUs. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021,*

Seoul, South Korea, February 27 - March 3, 2021, Jae W. Lee, Mary Lou Soffa, and Ayal Zaks (Eds.). IEEE, 27–38. https://doi.org/10.1109/CGO51591.2021.9370337

Wen Kokke, Jeremy G. Siek, and Philip Wadler. 2020. Programming language foundations in Agda. *Sci. Comput. Program.* 194 (2020), 102440. https://doi.org/10.1016/J.SCICO.2020.102440

Smail Kourta, Adel Abderahmane Namani, Fatima Benbouzid-Si Tayeb, Kim M. Hazelwood, Chris Cummins, Hugh Leather, and Riyadh Baghdadi. 2022. Caviar: an e-graph based TRS for automatic code optimization. In *CC '22: 31st ACM SIGPLAN International Conference on Compiler Construction, Seoul, South Korea, April 2 - 3, 2022*, Bernhard Egger and Aaron Smith (Eds.). ACM, 54–64. https://doi.org/10.1145/3497776.3517781

Lukasz Lachowski. 2018. On the Complexity of the Standard Translation of Lambda Calculus into Combinatory Logic. *Reports Math. Log.* 53 (2018), 19–42. https://rml.tcs.uj.edu.pl/rml-53/02-lachowski.pdf

Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 75–88. https://doi.org/10.1109/CGO.2004.1281665

Florian Lemaitre, Benjamin Couturier, and Lionel Lacassagne. 2017. Cholesky factorization on SIMD multi-core architectures. *J. Syst. Archit.* 79 (2017), 1–15. https://doi.org/10.1016/J.SYSARC.2017.06.005

A Solar Lezama. 2008. *Program synthesis by sketching.* Ph. D. Dissertation. PhD thesis, EECS Department, University of California, Berkeley.

Jannis Limperg and Asta Halkjær From. 2023. Aesop: White-Box Best-First Proof Search for Lean. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*, Robbert Krebbers, Dmitriy Traytel, Brigitte Pientka, and Steve Zdancewic (Eds.). ACM, 253–266. https://doi.org/10.1145/3573105.3575671

Saeed Maleki, Yaoqing Gao, María Jesús Garzarán, Tommy Wong, and David A. Padua. 2011. An Evaluation of Vectorizing Compilers. In *2011 International Conference on Parallel Architectures and Compilation Techniques, PACT 2011, Galveston, TX, USA, October 10-14, 2011*, Lawrence Rauchwerger and Vivek Sarkar (Eds.). IEEE Computer Society, 372–382. https://doi.org/10.1109/PACT.2011.68

Naums Mogers, Valentin Radu, Lu Li, Jack Turner, Michael F. P. O'Boyle, and Christophe Dubach. 2020. Automatic generation of specialized direct convolutions for mobile GPUs. In *GPGPU@PPoPP '20: 13th Annual Workshop on General Purpose Processing using Graphics Processing Unit colocated with 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 23, 2020*, Adwait Jog, Onur Kayiran, and Ashutosh Pattnaik (Eds.). ACM, 41–50. https://doi.org/10.1145/3366428.3380771

Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic Optimization for Image Processing Pipelines. (2015), 429–443. https://doi.org/10.1145/2694344.2694364

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 International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 31–44. https://doi.org/10.1145/3385412.3386012

Chandrakana Nandi, Max Willsey, Amy Zhu, Yisu Remy Wang, Brett Saiki, Adam Anderson, Adriana Schulz, Dan Grossman, and Zachary Tatlock. 2021. Rewrite rule inference using equality saturation. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–28. https://doi.org/10.1145/3485496

Charles Gregory Nelson. 1980. *Techniques for program verification.* Stanford University.

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

Robert Nieuwenhuis and Albert Oliveras. 2005. Proof-Producing Congruence Closure. In *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3467)*, Jürgen Giesl (Ed.). Springer, 453–468. https://doi.org/10.1007/978-3-540-32033-3_33

Robert Nieuwenhuis and Albert Oliveras. 2007. Fast congruence closure and extensions. *Inf. Comput.* 205, 4 (2007), 557–580. https://doi.org/10.1016/J.IC.2006.08.009

Jarkko Niittylahti, Juha Lemmetti, and Juhana Helovuo. 2002. High-performance implementation of wavelet algorithms on a standard PC. *Microprocess. Microsystems* 26, 4 (2002), 173–179. https://doi.org/10.1016/S0141-9331(02)00011-X

Tobias Nipkow. 2002. Structured Proofs in Isar/HOL. In *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers (Lecture Notes in Computer Science, Vol. 2646)*, Herman Geuvers and Freek Wiedijk (Eds.). Springer, 259–278. https://doi.org/10.1007/3-540-39185-1_15

David Parello, Olivier Temam, Albert Cohen, and Jean-Marie Verdun. 2004. Towards a Systematic, Pragmatic and Architecture-Aware Program Optimization Process for Complex Processors. In *Proceedings of the ACM/IEEE SC2004 Conference on High Performance Networking and Computing, 6-12 November 2004, Pittsburgh, PA, USA, CD-Rom*. IEEE Computer Society, 15. https://doi.org/10.1109/SC.2004.61

Lawrence C. Paulson and Kong Woei Susanto. 2007. Source-Level Proof Reconstruction for Interactive Theorem Proving. In *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4732)*, Klaus Schneider and Jens Brandt (Eds.). Springer, 232–245. https://doi.org/10.1007/978-3-540-74591-4_18

Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1384)*, Bernhard Steffen (Ed.). Springer, 151–166. https://doi.org/10.1007/BFB0054170

Varot Premtoon, James Koppel, and Armando Solar-Lezama. 2020. Semantic code search via equational reasoning. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 1066–1082. https://doi.org/10.1145/3385412.3386001

Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe, and Frédo Durand. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.* 31, 4 (2012), 32:1–32:12. https://doi.org/10.1145/2185520.2185528

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. (2013), 519–530. https://doi.org/10.1145/2491956.2462176

Alexandre Riazanov and Andrei Voronkov. 1999. Vampire. 1632 (1999), 292–296. https://doi.org/10.1007/3-540-48660-7_26

Joseph J Rotman. 2006. *A first course in abstract algebra: with applications.*

Peter Scholze. 2021. Liquid tensor experiment. *Experimental Mathematics* (2021), 1–6.

Daniel Selsam and Leonardo de Moura. 2016. Congruence Closure in Intensional Type Theory. In *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9706)*, Nicola Olivetti and Ashish Tiwari (Eds.). Springer, 99–115. https://doi.org/10.1007/978-3-319-40229-1_8

Savvas Sioutas, Sander Stuijk, Twan Basten, Henk Corporaal, and Lou J. Somers. 2020. Schedule Synthesis for Halide Pipelines on GPUs. *ACM Trans. Archit. Code Optim.* 17, 3 (2020), 23:1–23:25. https://doi.org/10.1145/3406117

Gus Henry Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael B. Taylor, Luis Ceze, and Zachary Tatlock. 2021. Pure tensor program rewriting via access patterns (representation pearl). (2021), 21–31. https://doi.org/10.1145/3460945.3464953

Mark Stephenson, Saman P. Amarasinghe, Martin C. Martin, and Una-May O'Reilly. 2003. Meta optimization: improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, Ron Cytron and Rajiv Gupta (Eds.). ACM, 77–90. https://doi.org/10.1145/781131.781141

Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, Kathleen Fisher and John H. Reppy (Eds.). ACM, 205–217. https://doi.org/10.1145/2784731.2784754

Michel Steuwer, Thomas Koehler, Bastian Köpcke, and Federico Pizzuti. 2022. RISE & Shine: Language-Oriented Compiler Design. *CoRR* abs/2201.03611 (2022). arXiv:2201.03611 https://arxiv.org/abs/2201.03611

Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: a functional data-parallel IR for high-performance GPU code generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, Vijay Janapa Reddi, Aaron Smith, and Lingjia Tang (Eds.). ACM, 74–85. http://dl.acm.org/citation.cfm?id=3049841

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 264–276. https://doi.org/10.1145/1480881.1480915

Sid Ahmed Ali Touati and Denis Barthou. 2006. On the decidability of phase ordering problem in optimizing compilation. In *Proceedings of the Third Conference on Computing Frontiers, 2006, Ischia, Italy, May 3-5, 2006*. ACM, 147–156. https://doi.org/10.1145/1128022.1128042

Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 2021. Vectorization for digital signal processors via equality saturation. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 874–886. https://doi.org/10.1145/3445814.3446707

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

13–26. https://doi.org/10.1145/289423.289425

Yisu Remy Wang, Shana Hutchison, Dan Suciu, Bill Howe, and Jonathan Leang. 2020. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *Proc. VLDB Endow.* 13, 11 (2020), 1919–1932. http://www.vldb.org/pvldb/vol13/p1919-wang.pdf

Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. 2009. SPASS Version 3.5. In *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5663)*, Renate A. Schmidt (Ed.). Springer, 140–145. https://doi.org/10.1007/978-3-642-02959-2_10

Freek Wiedijk. 2003. Formal Proof Sketches. In *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 3085)*, Stefano Berardi, Mario Coppo, and Ferruccio Damiani (Eds.). Springer, 378–393. https://doi.org/10.1007/978-3-540-24849-1_24

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 (2021), 1–29. https://doi.org/10.1145/3434304

Chenming Wu, Haisen Zhao, Chandrakana Nandi, Jeffrey I. Lipton, Zachary Tatlock, and Adriana Schulz. 2019. Carpentry compiler. *ACM Trans. Graph.* 38, 6 (2019), 195:1–195:14. https://doi.org/10.1145/3355089.3356518

Yichen Yang, Phitchaya Mangpo Phothilimthana, Yisu Remy Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. (2021). https://proceedings.mlsys.org/paper/2021/hash/65ded5353c5ee48d0b7d48c591b8f430-Abstract.html

Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 863–879. https://www.usenix.org/conference/osdi20/presentation/zheng

Oleksandr Zinenko. 2016. *Interactive Program Restructuring. (Restructuration interactive des programmes)*. Ph. D. Dissertation. University of Paris-Saclay, France. https://tel.archives-ouvertes.fr/tel-01414770