Sketch-Guided Equality Saturation

Scaling Equality Saturation to Complex Optimizations in Languages with Bindings

Thomas Kœhler
University of Glasgow
Scotland, UK
thomas.koehler@thok.eu

Phil Trinder
University of Glasgow
Scotland, UK
Phil.Trinder@glasgow.ac.uk

Michel Steuwer
University of Edinburgh
Scotland, UK
michel.steuwer@ed.ac.uk

Abstract

Equality saturation is a technique for implementing rewrite-driven compiler optimizations by efficiently representing many equivalent programs in so-called e-graphs. To improve performance, the set of equivalent programs is grown by applying rewrites in a purely additive way until a fixed point is reached (saturation), or the search times out.

In practice, two issues limit the application of equality saturation in programming language compilers. First, equality saturation is not efficient for the name bindings (variables) that appear in almost all programming languages. Second, equality saturation does not scale to complex optimizations with long rewrite sequences such as loop blocking.

This paper addresses both issues, thereby enabling equality saturation to be applied to more realistic programs and compiler optimizations. First, we demonstrate how to drastically improve the efficiency of equality saturation for a functional language based on the typed lambda calculus. Second, we introduce *sketch-guided equality saturation*, a semi-automatic technique that allows programmers to provide sketches guiding rewriting when performing complex optimizations.

We evaluate sketch-guided equality saturation by performing a series of realistic optimizations of matrix multiplication expressed in the Rise functional language. The optimizations include loop blocking, vectorization, and parallelization. We demonstrate that naive equality saturation does not scale to these optimizations, even with hours of exploration time. Previous work on orchestrating rewrite sequences shows that these optimizations can be expressed as rewrites, at the cost of weeks of programmer effort. Our guided equality saturation combines the advantages of both techniques: minimal programmer guidance enables complex compiler optimizations to be applied in seconds.

1 Introduction

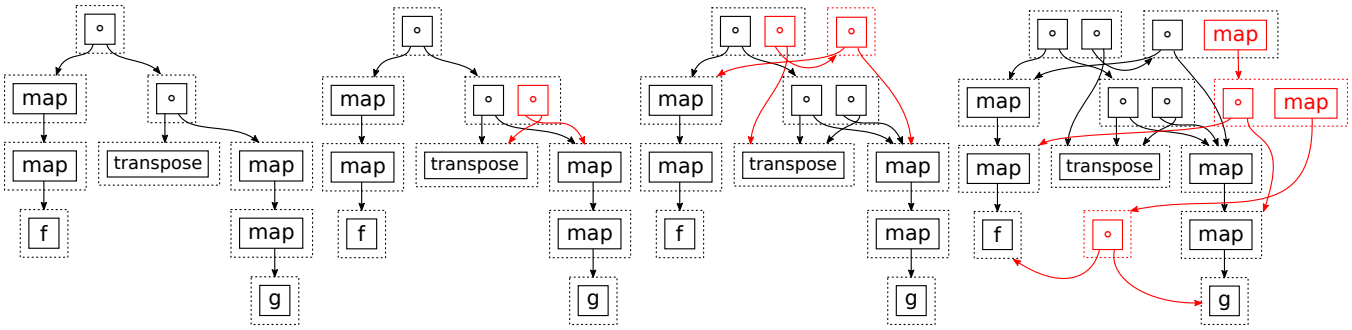
Term rewriting has been effective in optimizing compilers for decades [7]. However, deciding when to apply each rewrite rule is hard and has a huge impact on the performance of the rewritten program: the so-called *phase ordering problem*. The challenge is that the global benefit of applying a rewrite rule depends on future rewrites. Maximizing local benefit in a greedy fashion is not sufficient in the absence of a convergence property, i.e. confluence and termination, as local optima may be far away from the global optimum.

Equality saturation [30, 34] mitigates the phase ordering problem by exploring many ways to apply rewrite rules. Starting from an input program, an equality graph (*e-graph*) is grown iteratively until reaching a fixed point (saturation), achieving a goal, or timing out. An e-graph efficiently represents a large set of equivalent programs, and is grown by repeatedly applying all possible rewrite rules in a purely additive way. 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.

The applicability of equality saturation has recently been broadened by introducing an amortized invariant restoration technique called rebuilding and a mechanism called e-class analyses [34]. Nevertheless, the application of equality saturation for complex optimizations of realistic programs is limited by the following two issues.

Languages with name bindings. Previous equality saturation work either explicitly avoids the use of name binding for efficiency reasons [27], or uses a simple but inefficient implementation [34]. As almost all programming languages use variables, and hence name binding, this paper explores practical ways of efficiently implementing equality saturation for languages with name bindings. We study equality saturation for the lambda calculus as it is the standard formalism for functional languages. We show that using De Bruijn indices avoids overloading the e-graph with α-equivalent terms, and that an approximate substitution enables searches performed in milliseconds where searches with naive explicit substitution quickly run out of memory.

Complex optimizations, i.e. those requiring long rewrite sequences. On each equality saturation iteration, the e-graph tends to grow bigger since every possible rewrite rule is applied in a purely additive way. The growth rate is extremely rapid for some combinations of rewrite rules, such as associativity and commutativity that generate an exponential number of equivalent permutations [20, 32, 34]. In such cases, discovering long rewrite sequences that require many iterations is unfeasible. One way to address this issue is to limit the number of rules applied [32, 34], but this risks not finding optimizations that require rules that have been omitted. A second way is to use an external solver to speculatively



(a) representing the initial term **(b)** after applying rewrite (1) **(c)** after applying rewrite (2) **(d)** after applying rewrite (3) twice

Figure 1. Growing 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.

add equivalences [20], but this requires the identification of sub-tasks that can benefit from being delegated.

This paper proposes *sketch-guided equality saturation* as a technique to break down complex optimizations into smaller ones. The programmer specifies rewrite goals by writing *sketches*: program patterns that leave details unspecified. While sketches have previously been used as a starting point for program synthesis [16], our work uses sketches to end an equality saturation search once the sketch is satisfied. Guiding the rewriting using a sequence of sketches decomposes it into a sequence of relatively small equality saturation searches.

We demonstrate that sketch-guiding enables complex optimizations in the Rise functional language. We start by showing that existing equality saturation techniques are not sufficient for applying complex optimizations as the e-graph grows too large. Previous work on Rise produced highly optimized code at the cost of the programmer orchestrating sequences of thousands of rewrite rules [10]. Our evaluation demonstrates that by combining our efficient name binding techniques with sketch-guiding, complex optimizations are discovered by equality saturation with little programmer guidance and in a matter of seconds.

To summarize, the contributions of this paper include:

- The development of new techniques to support efficient equality saturation for a typed lambda calculus. The techniques are realized in the Risegg implementation for the Rise data-parallel functional language. We demonstrate the effectiveness of Risegg by optimizing a binomial filter application (section 3).
- Proposing *sketch-guided equality saturation* as a new semi-automatic technique to perform complex optimizations that require long rewrite sequences not discoverable by equality saturation alone. We demonstrate the practicality of sketches for guiding realistic optimizations of Harris corner detection (section 4).
- A systematic comparative evaluation of sketch-guided equality saturation for optimizing matrix multiplication. Seven complex optimizations are demonstrated, including loop blocking, vectorization, and parallelization. We show that the complex optimizations are not feasible with fully automated equality saturation due to excessive runtime and memory consumption. In contrast, sketch-guided equality saturation performs the optimizations in seconds and with low memory consumption. At most four sketches are required to guide the search, i.e. significantly less effort than purely manual techniques (section 5).

2 Background

This section gives a technical overview of equality saturation and its application to express compiler optimizations. We also introduce the functional language Rise [10] in which the programs we optimize in this paper are expressed.

2.1 Equality saturation

Equality saturation [30, 34] is a technique for efficiently implementing rewrite-driven compiler optimizations without committing to a single rewrite choice. We demonstrate how equality saturation mitigates the phase ordering problem by using a rewriting example where greedily reducing a cost function is not sufficient to find the optimal program.

Rewriting is often used to fuse operators and avoid that every operator writes its result to memory, for example:

$$(map\ (map\ f)) \circ (transpose \circ (map\ (map\ g))) \quad \text{(a)}$$

$$\longrightarrow^{*}$$

$$(map\ (map\ (f \circ g))) \circ transpose \quad \text{(b)}$$

The initial term (a) applies function g to each element of a matrix (using two nested *map*s), transposes the result, and then applies function f to each element. The optimized term

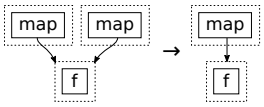


Figure 2. The congruence invariant simplifies the e-graph on the left by merging two identical e-nodes for *map f* into a single e-node as shown on the right.

(b) avoids storing an intermediate matrix in memory and transposes the input before applying g and f to each element. The following rewrite rules are sufficient to perform this optimization, if applied in the correct order:

$$transpose \circ map\,(map\,a) \longleftrightarrow map\,(map\,a) \circ transpose \quad (1)$$

$$a \circ (b \circ c) \longleftrightarrow (a \circ b) \circ c \quad (2)$$

$$map\ a \circ map\ b \longrightarrow map\ (a \circ b) \quad (3)$$

Rule (1) states that transposing a two-dimensional array before or after applying a function to the elements is equivalent. Rule (2) states that function composition is associative. Finally, rule (3) is the rewrite rule for map fusion. In this example, minimizing the term size results in maximizing fused maps and, therefore, is a good cost model.

If we greedily apply rewrite rules that lower term size, we will only apply rule (3) as this is the only rule that reduces term size. However, rule (3) cannot be directly applied to term (a): we are in 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 now investigate step-by-step how equality saturation enables to minimize term size by exploring many possible ways to apply rewrites without getting stuck in local minima.

First, an equality graph (e-graph) representing the initial term is constructed (fig. 1a). 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 n child e-classes (e_i). Examples of symbols are *map*, *transpose*, and $\circ$. The e-graph data structure is used during equality saturation to efficiently represent and rewrite a set of equivalent programs.

Second, the e-graph is iteratively grown by applying rules non-destructively (figs. 1b to 1d). While standard term rewriting picks a single possible rewrite in a depth-first manner, equality saturation explores all possible rewrites in a breadth-first manner. Within an equality saturation iteration, rewrites are applied independently: they may only depend on rewrites from previous iterations. For the sake of simplicity, we only apply a handful of rewrite rules in fig. 1. When applying a rewrite rule, the equality between its left-hand side and its right-hand side is recorded in the e-graph. Rewrite rules stop being applied when a fixed point is reached (saturation), or when another stopping criteria is reached (e.g. timeout). If saturation is reached, it means that all possible rewrites have been explored.

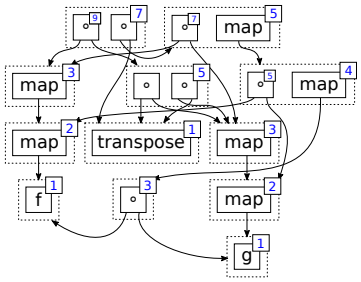


Figure 3. Smallest term size computed for each e-class using an e-class analysis (blue number in the top-right corners of e-classes). For illustrative purposes, we also show the smallest term size for e-nodes where it differs from its e-class value.

At that point, the e-graph represents many terms that are equivalent according to the applied rules. An e-graph is much more compact than a regular set of terms, as equivalent sub-terms are shared. E-graphs are capable of representing exponentially many terms in polynomial space, and even infinitely many terms in the presence of cycles [34]. To maximize sharing, a *congruence invariant* is maintained: intuitively identical e-nodes should not be in different e-classes (fig. 2). However, we will see later that extensive sharing does not necessarily prevent the e-graph size from exploding.

Finally, an *extraction* procedure selects the best term from the e-graph according to a cost function. For a *local cost function* c (i.e. with signature $c(F(k_1 : K, .., k_n : K)) : K$ if the cost is of type K), a relatively simple bottom-up e-graph traversal can be used [23]. More complex cost functions require more complex extraction procedures [32, 35].

An *e-class analysis* [34] enables propagating an analysis data of type D in a bottom-up fashion, and can be used for extraction when the cost function is local. An e-class analysis is defined by providing two functions:

- A function constructing the analysis data from an n-ary symbol F combined with the data d_i of its child e-classes:

$$make(F(d_1 : D, .., d_n : D)) : D$$

- A function merging the analysis data of e-nodes that are in the same e-class:

$$merge(d_1 : D, d_2 : D) : D$$

To compute the smallest term for each e-class, we define an e-class analysis with $D = (\text{term size}, \text{term})$:

$$make(F(d_1, .., d_n)) = (1 + \sum_i fst(d_i), F(snd(d_1), .., snd(d_n)))$$

$$merge(d_1, d_2) = \text{if } fst(d_1) \leq fst(d_2) \text{ then } d_1 \text{ else } d_2$$

The resulting term size component of the analysis is shown in fig. 3. It now only requires looking at the analysis data for the e-class of term (a) to extract term (b) as the optimized term. The optimized term is of size 7 compared to the initial term of size 9.

2.2 Rewriting the Rise functional language

In this paper we study the lambda calculus because it formalizes functional languages. To demonstrate the impact of our approach in practice, we use Rise [10] that implements a typed lambda calculus, and many of the examples in this paper are Rise programs. Rise is a spiritual successor of Lift [28, 29] that demonstrated performance portability across hardware by automatically applying semantics-preserving rewrite rules to optimize programs from domains including scientific code [11] and convolutions [18].

Rise provides standard lambda abstraction (`λx. b`), function application (`f x`), identifiers and literals. Rise also offers an extensible set of higher-order functions describing well-known data-parallel computational patterns. While Rise provides many computational patterns, we focus in this paper on two important patterns. `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. To make Rise accessible to equality saturation, Rise programs are easily encoded as terms of shape $F(t_1, .., t_n)$ as shown in table 1.

To control optimization Rise is complemented by a second programming language, Elevate [10] that allows programmers to describe complex compiler optimizations as compositions of rewrite rules, called *strategies*. The performance of the code generated by Rise and Elevate has been to shown to be on par with the state-of-the-art deep learning compiler TVM [3] for matrix multiplication [10]; and competitive to - or even up to 1.4× better than - the state-of-the-art image processing compiler Halide [26] for the Harris corner detection [14]. This makes Rise an interesting base language for exploring rewrite-based compiler optimizations.

Unfortunately writing Elevate strategies manually is low level and time-consuming. A strategy describes precisely the rewrite sequence required for a particular optimization. Even though Elevate provides combinators and abstractions to help express complex optimizations, the authors of [10] and [14] report that expressing complex optimizations required between 2 and 4 person weeks of effort. The fundamental problem is that Elevate strategies express optimizations in an imperative style and require the programmer to orchestrate all rewrite steps deterministically. Besides being costly to develop, this also significantly limits applicability of optimizations to many different programs: small program differences require adjustments to the rewrite sequence. So it is highly desirable to find some automatic, or at least semi-automatic, rewriting technique to reduce the programmer effort required to optimize Rise programs.

Rise program	F	$t_1, .., t_n$
`λx. b`	lam x	b
`f x`	app	f, x
`x`	var x	
`map`	map	
`reduce`	reduce	

Table 1. Correspondence between Rise programs and $F(t_1, .., t_n)$ term representation for the purposes of equality saturation.

$$(\lambda x.\ b)\ e \longrightarrow b[e/x] \qquad (\beta\text{-reduction})$$

$$\lambda x.\ f\ x \longrightarrow f \qquad (\eta\text{-reduction})$$
$$\text{if } x \text{ not free in } f$$

$$map\ f\ (map\ g\ arg) \longrightarrow map\ (\lambda x.\ f\ (g\ x))\ arg \qquad \text{(map-fusion)}$$

$$map\ (\lambda x.\ f\ gx) \longrightarrow \lambda y.\ map\ f\ (map\ (\lambda x.\ gx)\ y) \qquad \text{(map-fission)}$$
$$\text{if } x \text{ not free in } f$$

Figure 4. Rise rewrite rules using substitution, name bindings (lambda abstractions), and freshness predicates.

3 Efficient Equality Saturation for the Lambda Calculus

This section addresses the first issue with prior equality saturation techniques: the lack of effective support for languages with name bindings. We explore the engineering design choices required to efficiently implement equality saturation for a typed lambda calculus. A set of design choices are realized for the Rise language in the new Risegg implementation that is heavily inspired by the egg library [34]. The performance numbers in this section are from a prototype of Risegg[1] for an untyped subset of Rise implemented in Rust on top of the egg library.

To assess the efficiency of equality saturation in this section, our aim is to be able to discover certain rewrite goals in reasonable time on a laptop machine, i.e. with an AMD Ryzen 5 PRO 2500U processor and using no more than 4GB of RAM. *Discovering a rewrite goal* means that it is feasible to grow an e-graph starting from the initial program until the goal program is represented in the e-class of the initial program.

Applying equality saturation to lambda calculus terms requires the efficient support of standard operations and rewrites. Figure 4 shows the standard rules of β-reduction and η-reduction. The other two rules encode standard map-fusion and map-fission, and are interesting because they introduce new name bindings on their right-hand-side.

3.1 Substitution

In equality saturation standard term substitution cannot be used to directly compute $b[e/x]$ from the β-reduction rule in an e-graph, because b and e are not terms but e-classes.

[1] https://github.com/Bastacyclop/egg-rise

A simple way to address this is to use *explicit substitution* as in egg [34]. A syntactic constructor is added to represent substitution, as well as rewrite rules to encode its small-step behavior:

$$(a\ b)[e/v] \longrightarrow (a[e/v]\ b[e/v])$$
$$v[e/v] \longrightarrow e$$

Explicit substitution adds all intermediate substitution steps to the e-graph, quickly exploding its size. We can demonstrate the e-graph size problem by attempting to discover a trivial rewrite goal using equality saturation:

$$\begin{aligned} &map\ (\lambda x.\ f_4\ (f_3\ (f_2\ (f_1\ x)))) \longrightarrow^* \\ &\lambda y.\ map\ (\lambda x.\ f_4\ (f_3\ x))\ (map\ (\lambda x.\ f_2\ (f_1\ x))\ y) \end{aligned} \tag{4}$$

The rewrite in (4) merely requires a sequence of two map-fission rules, one map-fusion rule, plus a couple of the η-reduction and β-reduction rules that are pervasive when rewriting functional programs:

$$\begin{aligned} &map\ (\lambda x.\ f_4\ (f_3\ (f_2\ (f_1\ x)))) \longrightarrow^* \\ &\lambda y.\ map\ f_4\ (map\ (\lambda x.\ f_3\ (f_2\ (f_1\ x)))\ y) \longrightarrow^* \\ &\lambda y.\ map\ f_4\ (map\ f_3\ (map\ (\lambda x.\ f_2\ (f_1\ x))\ y) \longrightarrow^* \\ &\lambda y.\ map\ (\lambda x.\ f_4\ (f_3\ x))\ (map\ (\lambda x.\ f_2\ (f_1\ x))\ y) \end{aligned} \tag{5}$$

Despite this simple rewrite sequence, rewrite goal (4) cannot reasonably be discovered using explicit substitution. After more than 40 seconds of equality saturation (10 iterations) the available 4GB memory is exhausted and the goal has not been discovered. The e-graph contains 13M e-nodes and 3M e-classes.

So intermediate substitution steps cannot be added to the e-graph, otherwise it grows uncontrollably. To avoid intermediate substitutions we propose *extraction-based substitution* that works as followings.

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.

Extraction-based substitution is far more efficient than explicit substitution. For example we can discover the rewrite goal (4) in less than a millisecond, with 5 iterations, and the e-graph contains only 364 e-nodes and 277 e-classes.

Extraction-based substitution is, however, an approximation as it computes the substitution for a subset of the terms represented by b and e, and ignores the rest. Figure 5 shows an example where the initial e-graph is in the middle, and the e-graph after extraction-based substitution with $b = x$ and $e = y$ on the right. This particular choice results in an e-graph lacking the *id x* program that is included in the e-graph without approximation (left in fig. 5).

In practice, we have not observed this approximation to be an issue, and believe this is for two reasons. First, the substitution is computed on each equality saturation iteration, where different terms may be extracted, increasing coverage

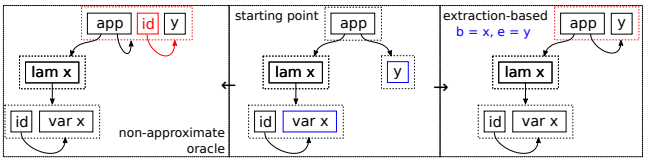


Figure 5. Example of β-reduction with extraction-based substitution (right). The initial e-graph (middle) represents $(\lambda x.\ id\ x)\ y$. After β-reduction, the e-graph does not represent *id y* because x has been extracted for b in the rewrite rule; ignoring *id x*.

```
def dot a b = reduce + 0 (map (λy. (fst y) × (snd y))
```

```
map (map λnbh. dot (join weights2d) (join nbh))
  (map transpose (slide 3 1 (map (slide 3 1) input)))
```

$$\longrightarrow^*$$

```
map (λnbhL.
    map (λnbhH. dot weightsH nbhH)
    (slide 3 1 (map (λnbhV. dot weightsV nbhV)
                transpose nbhL)))
  (slide 3 1 input)
```

Figure 6. Optimizing the binomial filter. The initial Rise program iterates over 2D neighborhoods (`nbh`). A `dot` product is computed between the weights (`weight2d` shown in eq. (6)) and each neighborhood. The optimized program iterates over two 1D neighborhoods (vertical and horizontal) instead.

of the set of terms represented by b and e. Second, many of the ignored equivalences are recovered either by e-graph congruence, or by applying further rewrite rules.

Future work may investigate alternative substitution implementations to balance efficiency with non-approximation. For efficiency extraction-based substitution is used in Risegg.

3.2 Name Bindings

In equality saturation inappropriate handling of name bindings easily leads to serious efficiency issues. Consider rewrite rules like map-fusion that create a new lambda abstraction on their right-hand side. Which name should they introduce when they are applied? In standard term rewriting, generating a fresh name using a global counter (aka. gensym) is a common solution. However, if a new name is generated each time the rewrite rule is applied, the e-graph will quickly be overloaded with many α-equivalent terms[2].

Fewer α-equivalent terms are introduced if fresh names are generated as a function of the matched e-class identifiers. However as the e-graph grows and e-classes are merged e-class identifiers change, and α-equivalent terms are still generated and duplicated in the e-graph.

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

Figure 6 shows an example that demonstrates the practical issues when rewriting with α-equivalent terms. This Rise program computes a binomial filter – a 2D convolution – that is expressed using the `slide` pattern creating a sliding window to group neighboring elements that are then multiplied with the convolution kernel and summed. The purpose of the rewrite shown is to separate the 2D filter into two 1D filters according to a well-known convolution kernel equation:

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 1 \end{bmatrix} \tag{6}$$

This separation optimization reduces both memory accesses and arithmetic complexity. An Elevate rewriting strategy achieves the optimization by orchestrating 30 rewrite rules including 17 η/β-reductions [14].

The binomial filter optimization goal cannot be discovered by equality saturation if fresh names are generated for each rewrite rule application. After two minutes and 9 iterations the 4GB memory is exhausted, and the goal has not been discovered. The e-graph contains 2.9M e-nodes and 1.4M e-classes, emphasizing the need for more efficient name handling.

De Bruijn indices [5] are a standard technique for representing lambda calculus terms without naming the bound variables, and avoid the need for α conversions. If De Bruijn indices enable two α-equivalent terms to become structurally equivalent, the regular e-graph congruence invariant[3] is enough to prevent the duplication of α-equivalent terms. Therefore, we translate our terms and rewrite rules to use De Bruijn indices instead of names, and observe a significant change in efficiency. With De Bruijn indices, 100ms is enough to discover the binomial filter rewrite goal. After 11 iterations, the e-graph contains 3K e-nodes and 1K e-classes. Hence De Bruijn indices are used in Risegg.

True equality modulo α-renaming. While De Bruijn indices give a significant performance improvement they do not provide equality modulo α-renaming for sub-terms. Consider $f\ (\lambda x.\ f) = \%0\ (\lambda.\ \%1)$, where $\%i$ represents De Bruijn indices. Although %0 and %1 are structurally different, they both correspond to the same variable f. Recent work has shown how to implement efficient hashing modulo alpha renaming [17], and could be used to investigate an even more efficient e-graph representation. Another possibility would be to investigate the effectiveness of nominal rewriting techniques [8].

Translating name-based rules into index-based rules. Using De Bruijn indices means that rewrite rules also need to manipulate terms with De Bruijn indices. Thankfully, more user-friendly name-based rewrite rules can be automatically translated to the index-based rules used internally [2].

[3] Reminder: the congruence invariant ensures that identical e-nodes will not end up in different e-classes.

Shifting De Bruijn indices. De Bruijn indices must be shifted when a term is used with different surrounding lambdas. In Risegg, *extraction-based index shifting* works as substitution in three steps:

- extract a term from the e-class whose indices need shifting;
- perform standard index shifting;
- add the resulting term to the e-graph.

Avoiding Name Bindings using Combinators. It is also possible to avoid name bindings entirely [27]. For example, it is possible to introduce a function composition combinator '$\circ$' (also used in section 2.1), that greatly simplifies map-fusion and map-fission rules:

$$f\ (g\ x) \longrightarrow (f \circ g)\ x \qquad (\circ\text{-intro})$$

$$map\ f \circ map\ g \longrightarrow map\ (f \circ g) \qquad (\text{map-fusion}_2)$$

$$map\ (f \circ g) \longrightarrow map\ f \circ map\ g \qquad (\text{map-fission}_2)$$

However, this approach has its own downsides. Associativity rules are required, which we know increases the growth rate of the e-graph [34]. Only using a left-/right-most associativity rule avoids generating too many equivalent ways to parenthesize terms. But other rewrite rules now have to take this associativity convention into account, making their definition more difficult and their matching more expensive. In general, matching modulo associativity or commutativity are algorithmically hard problems [1].

The $\circ$ combinator on its own is also not sufficient to remove the need for name bindings. At one extreme, combinatory logic could be used as any lambda calculus term can be represented in combinatory logic, replacing function abstraction by a limited set of combinators. However, translating a lambda calculus term into combinatory logic results in a term of size $O(n^3)$ in the worst case, where n is the size of the initial term [15]. Translating existing rewrite systems to combinatory logic would be challenging in itself.

3.3 Freshness Predicates

Handling predicates is not trivial in equality saturation. The η-reduction has the side condition that "if x not free in f", but in an e-graph f is an e-class and not a term.

The predicate could be handled precisely by filtering the e-class f into $f' = \{t \mid t \in f, x \text{ not free in } t\}$, and using f' on the right-hand-side of the rule. However, this requires splitting an e-class in two: one that satisfies the predicate, and one that does not. This would reduce sharing and increase the e-graph size, be difficult to reason about in the presence of cycles, and interfere with the e-graph amortized invariant restoration optimization [34].

The design of Risegg makes an engineering trade-off, as for substitution, and following egg [34]. In Risegg, the η-reduction rewrite rule is only applied if $\forall t \in f.\ x$ not free in t. Advantages are that this predicate is efficient to compute using an e-class analysis, and that there is no need to split the e-class. The disadvantage is that it is an approximation

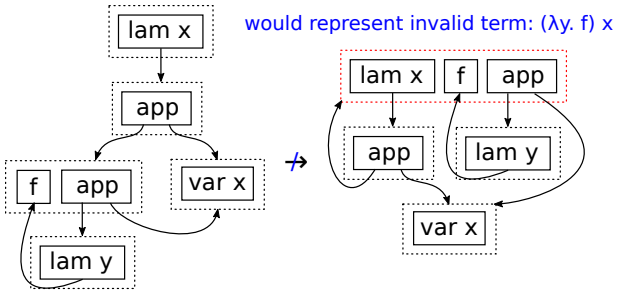


Figure 7. Example where η-reduction is not applied. The initial e-graph represents $\lambda x.\ f\ x$, but η-reduction is not triggered because $f = (\lambda y.\ f)\ x$ where x is free. Using $\exists$ instead of $\forall$ in our predicate, we would obtain the e-graph on the right which represents f, but also invalid terms such as $(\lambda y.\ f)\ x$ where x is not bound anymore.

that ignores some valid terms. Figure 7 shows an example where η-reduction is not applied by Risegg. In practice, we have not observed this approximation to be an issue, e.g. for the results presented in section 5.

3.4 Adding Types

Typed lambda calculi are pervasive providing the foundation for almost all functional languages, and a key consideration is how to add types into the e-graph. Considering the terms $(\lambda x.\ x)\ (0 : int)$ and $(\lambda x.\ x)\ (0.0 : float)$, there are broadly two alternatives:

- Keep types polymorphic (one e-class):

$$\lambda x.\ x : \forall t.\ t \to t$$

- Instantiate the types (two e-classes):

$$\lambda x.\ x : \text{int} \to \text{int} \quad \neq \quad \lambda x.\ x : \text{float} \to \text{float}$$

While keeping types polymorphic enables more sharing, instantiating types enables more precise type-based rewriting. While polymorphic types can be computed using an e-class analysis, instantiated types must be embedded in the e-graph. There is no obvious best solution, instead we observe a trade-off between the amount of sharing and the amount of information. Since Rise rewrite rules often match precise types, types are instantiated in Risegg.

Rewrite rule type inference. To avoid having to explicitly type rewrite rules by hand, we infer their types. After inferring the types on the left-hand-side, we check that the right-hand-side is well-typed for any well-typed left-hand-side matching program. When applied, typed rewrite rules match (deconstruct) types with their left-hand-side, and construct types on their right-hand-side. Type annotations can be used in Risegg to constrain the inferred types.

Hash-consing types. Since types are duplicated many times in the e-graph, and since structural type equality is often required, we hash-cons types for efficiency [9].

3.5 Summary

This section has showed important aspects to consider when using equality saturation for languages with name bindings. Specifically, we have seen that De Bruijn indices and extraction-based substitution are critical in practice: making the difference between running out of memory or optimizing functional programs in seconds.

4 Sketch-Guided Equality Saturation

The previous section discussed crucial techniques for efficiently optimizing functional programs using equality saturation. But are these techniques enough to perform complex optimizations, i.e. those requiring long rewrite sequences? In [10] the authors report that 63,000 rewrite steps are required to perform loop blocking, vectorization, and parallelization optimizations for a matrix multiplication. Our evaluation in section 5 shows that even with the techniques discussed in section 3 equality saturation is unable to perform these optimizations, exhausting the available 60GB memory after more than one hour of search.

The issue is that as the e-graph grows, iterations become slower and require more memory. Combined with the fact that the e-graph is grown in a breadth-first manner, this makes finding long rewrite sequences inherently hard. The growth rate is aggravated by some combinations of rewrite rules, such as associativity and commutativity that generate an exponential number of equivalent permutations [20, 32, 34]. This motivates keeping the size of the e-graph small.

4.1 Keeping the e-graph size under control

Amongst the many ways to reduce e-graph size during exploration, we identify the following three general directions.

Deleting programs that are not considered valuable. Often there are programs that we are obviously not interested in and, therefore, these could be removed from the e-graph. For example, we implement a filter removing all e-classes that only contribute to constructing programs of more than a given size limit. This is useful because exploring unreasonably large terms is typically not desirable.

Unfortunately, due to the extensive sharing, deleting individual programs from the e-graph is difficult. Therefore, Risegg only deletes e-nodes and e-classes, affecting all of their parents. This prohibits deleting individual programs that share sub terms with other programs that we do not want to delete. In the future, it would be interesting to develop more sophisticated capabilities, e.g. deleting the matched left-hand-side of a rewrite rule that would also allow performing normalization.

Prioritizing growth in promising directions. Previous work proposes rewrite rule schedulers as a way to control which rewrite rules should be applied on a given equality saturation iteration [34]. The `SimpleScheduler` from the egg

```
slide(p+4, p) ▷ mapGlobal(
  circularBuffer(global, 3, grayLine) ▷
  circularBuffer(global, 3, sobelLine) ▷
  mapSeq(coarsityLine)
) ▷ join
```

Listing 1. Harris shape after circular buffering [14]

```
? ▷ isSlide(p+4, p) ▷ app(mapGlobal, contains(? ▷
 isCircularBuffer(global, 3, containsGrayLine) ▷
 isCircularBuffer(global, 3, containsSobelLine) ▷
 app(mapSeq, containsCoarsityLine)
))
```

Listing 2. Harris sketch after circular buffering

Figure 8. An abstract program snippet describing how the shape of a Rise Harris corner detection evolves after optimizations (listing 1). These shape intuitions can be captured by defining a sketch (listing 2) that features program holes (`?`) and constraints (`contains`).

library applies all rewrite rules on each iteration. The default `BackoffScheduler` prevents specific rules from being applied too often, reducing e-graph growth in the presence of "explosive" rules such as associativity and commutativity. Our experience with Rise is that using the `BackoffScheduler` is counterproductive because the desired optimizations depend on explosive rules. Future work may look into finding more advanced ways to prioritize e-graph growth, but at present Risegg does not use a rewrite rule scheduler.

Starting over. An effective way to reduce e-graph size is to build and grow a new e-graph from the most promising term represented in a previous e-graph. Our sketch-guided technique develops this approach by defining a sequence of equality saturation searches, while defining a clear goal for each search in the form of a *sketch*.

4.2 The Intuition for Sketches

When designing optimizations it is useful for the programmer to think about the desired *shape* of the optimized program. *Sketches* are program patterns that capture this intuition while leaving details unspecified.

Previous work on optimizing the Harris corner detection image processing pipeline with rewrites [14] used program snippets like listing 1 to explain the effect of the various optimizations. The *key insight* is that explanatory program snippets can be utilized as sketches such as the one shown in listing 2. This sketch resembles the snippet in listing 1 but adds program holes (`?`) and constraints (`contains`) to explicitly elide program details.

Where Elevate rewriting is purely manual, sketch-guided equality saturation is semi-automated. That is, it allows the programmer to declaratively specify the desired optimization goal (the sketch) without needing to specify the detailed rewrite sequence to get there.

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

$$\begin{aligned} R(?) &= T = \{F(t_1, .., t_n)\} \\ R(F(s_1, .., s_n)) &= \{F(t_1, .., t_n) \mid t_i \in R(s_i)\} \\ R(contains(s)) &= R(s) \cup \{F(t_1, .., t_n) \mid \exists t_i \in R(contains(s))\} \\ R(s_1 \vee s_2) &= R(s_1) \cup R(s_2) \end{aligned}$$

Figure 9. Grammar of SketchBasic (top) and terms represented by SketchBasic (bottom).

4.3 Sketch Definition

We define the simple SketchBasic language with just four constructors as a proof of concept. The syntax of SketchBasic and the set of terms that the constructors represent is defined in fig. 9. A sketch s represents a set of terms $R(s)$, such that $R(s) \subset T$ where T denotes all terms in the language we rewrite. We say that any $t \in R(s)$ satisfies the sketch s.

The ? sketch is the least precise as it represents 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. Finally, the $s_1 \vee s_2$ sketch represents terms satisfying either s_1 or s_2.

When rewriting terms in a typed language sketches may be annotated with a type sketch ($s :: pt$) constraining the type of terms. If $R(pt)$ denotes the set of terms satisfying the type sketch pt, then $R(s :: pt) = R(s) \cap R(pt)$. The grammar of type sketches depends on the language we rewrite. We elide type sketches from our definition of SketchBasic for simplicity, but use them in section 5.

When writing sketches, a balance has to be found between being too precise (representing only one program) and too vague (representing programs that are not desired). This balance also interacts with the choice of rules, since programs that may be found by the search are $R(s) \cap EQ(t, rules)$ where $EQ(t, rules)$ 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.

4.4 Sketch-Guided Equality Saturation Algorithm

The idea behind *Sketch-Guided Equality Saturation* is to use multiple sketches to decompose a complex rewrite goal into a series of simpler rewrite goals. The overall process is illustrated in fig. 10.

The programmer guides the system by providing a sequence of sketches ($sketch_1$, .., $sketch_n$). Successive equality saturation searches are performed to find equivalent terms satisfying one sketch after the other. Because each sketch

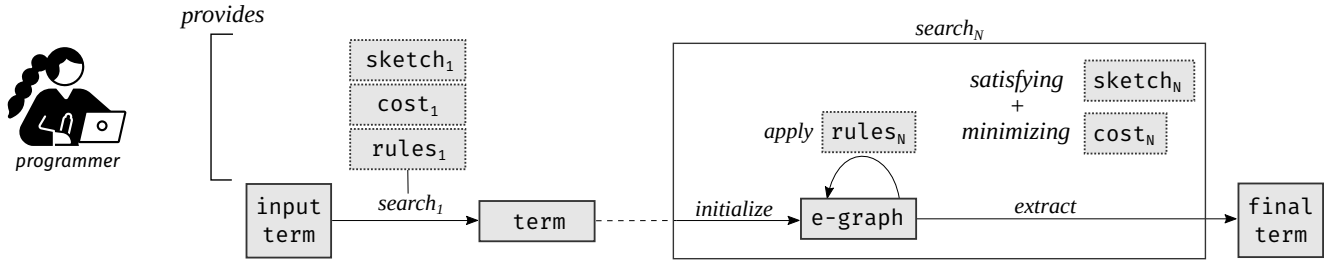


Figure 10. Diagram illustrating Sketch-Guided Equality Saturation. Starting from an input term, N-many consecutive searches are performed to find a term satisfying each sketch. All searches behave as detailed on the right for the N-th search.

is satisfied by many terms, the programmer must also provide cost models ($cost_1, .., cost_n$) to select the term to be used as the start point for the next search. Sets of rewrite rules ($rules_1, .., rules_n$) are provided to grow the e-graph in each search.

The pseudo-code for the sketch-guided equality saturation algorithm is shown in listing 3. The `extract` function (line 12) is used to extract a term from the e-graph that satisfies the specified sketch while minimizing the specified cost model, and we describe it in the next subsection. The `found` function (line 11) is used to stop growing the e-graph by checking whether `extract` succeeded in returning a term. For efficiency our implementation of `found` does not rely on `extract` and checks only if a term could be extracted.

4.5 Sketch-Guided Equality Saturation Extraction

To `extract` the best program that satisfies a SketchBasic sketch s from an e-graph g we define a helper function $E(c, s, g)$, where c is a cost function that must be monotonic and local. While `extract` returns a program from an e-class e, the helper E returns a map from e-classes to optional tuples of costs and terms. After invoking E we simply look up the e-class e in the map and extract the term from the optional tuple. For efficiency, we memoize previously computed results of E. The `extract` function is recursively defined over the four SketchBasic cases as follows.

Case 1: $s = ?$. This case is equivalent to extracting the programs minimizing c from the e-graph. We implement this extraction as an e-class analysis (section 2) with data

```
1  def SGES(term, params): Option[Term] =
2    if params.isEmpty
3    then Some(term)
4    else
5      (sketch, cost, rules) = params.head
6      g = create empty e-graph
7      normTerm = normalize(term)
8        using a configurable normal form
9      e = g.add(normTerm)
10     grow g using rules until found(g, e, sketch)
11     if found(g, e, sketch) then
12       nextTerm = extract(g, e, sketch, cost)
13       SGES(nextTerm, params.tail)
14     else
15       None
```

Listing 3. Sketch-guided Equality Saturation Algorithm

type $D = \text{Option}[(k, t)]$ and functions *make* that constructs analysis data and *merge* that combines analysis data from e-nodes in the same e-class:

$$make(F(d_1, .., d_n)) = \begin{cases} Some\left(\begin{array}{l} c(F(k_1, .., k_n)), \\ F(t_1, .., t_n) \end{array}\right) & (k_i, t_i) \in d_i \\ None & \text{otherwise} \end{cases}$$

$$merge(d_1, d_2) = \begin{cases} \text{if } k_1 \leq k_2 \text{ then } d_1 \text{ else } d_2 & (k_i, \_) \in d_i \\ d_1 & (k_1, \_) \in d_1 \\ d_2 & (k_2, \_) \in d_2 \\ None & \text{otherwise} \end{cases}$$

Case 2: $s = F(s_1, .., s_n)$. We consider each e-class e containing $F(e_1, .., e_n)$ e-nodes and the terms that should be extracted for each child e-class e_i. We write $E(c, s, g)[e]$ for indexing into the map returned by E:

$$E(c, F(s_1, .., s_n), g)[e] = \begin{cases} Some(c(F(k_1, .., k_n)), F(t_1, .., t_n)) & (k_i, t_i) \in E(c, s_i, g)[e_i] \\ None & \text{otherwise} \end{cases}$$

Case 3: $s = contains(s_2)$. We use another e-class analysis and initialize the analysis data to $E(c, s_2, g)$ corresponding to the base case where $R(s_2) \subset R(contains(s_2))$. To *make* the analysis data we consider all terms that would contain terms from s_2 and keep the best by reducing them using *merge*:

$$\begin{aligned} make(F(d_1, .., d_n)) = reduce\ merge \\ \{Some(c(F(k_1, .., k_j, .., k_n)), F(t_1, .., t_j, .., t_n)) \mid \\ i \neq j, (k_i, t_i) \in E(c, ?, g)[e_i], (k_j, t_j) \in d_j\} \end{aligned}$$

To *merge* the analysis data, we do the same as for $s = ?$.

Case 4: $s = s_1 \vee s_2$. We *merge* the results from s_1 and s_2:

$$E(c, s_1 \vee s_2, g)(e) = merge(E(c, s_1, g)(e), E(c, s_2, g)(e))$$

4.6 Summary

This section introduces sketch-guided equality saturation as a semi-automated process where the programmer guides multiple equality saturation searches. They do so by specifying sketches that declaratively describe the program shapes after the desired optimizations have been applied.

version	sketches	found	time	RAM	rules	e-graph
baseline	1	yes	0.5s	20 MB	2	51, 49
blocking	1	yes	1h+	35GB	5M	4M, 2M
vectorization	1	**no**	1h+	60GB+	???	???
loop-perm	1	**no**	1h+	60GB+	???	???
array-packing	1	**no**	35mn	60GB+	???	???
cache-blocks	1	**no**	35mn	60GB+	???	???
parallel	1	**no**	35mn	60GB+	???	???

Table 2. Optimization time and memory consumption for **fully automated equality saturation**. Only the "baseline" and "blocking" optimized versions could be found. All other optimizations could **not** be found, exceeding 60GB.

version	sketches	found	time	RAM	rules	e-graph
baseline	1	yes	0.5s	20 MB	2	51, 49
blocking	2	yes	7s	0.3 GB	11K	11K, 7K
vectorization	3	yes	7s	0.4 GB	11K	11K, 7K
loop-perm	3	yes	4s	0.3 GB	6K	10K, 7K
array-packing	4	yes	5s	0.4 GB	9K	10K, 7K
cache-blocks	4	yes	5s	0.5 GB	9K	10K, 7K
parallel	4	yes	5s	0.4 GB	9K	10K, 7K

Table 3. Optimization time and memory consumption for **sketch-guided equality saturation**. All optimizations are found in seconds using less than 0.5 GB of memory, requiring only up to 4 sketches.

5 Evaluation

This section compares three different optimization methods: Elevate rewriting strategies, fully automated equality saturation, and the new sketch-guided equality saturation. In the evaluation *both* equality saturation techniques use the efficient implementation of name bindings presented in section 3.

We evaluate the optimization of a matrix multiplication as it allows us to compare sketch-guided equality saturation against published Elevate strategies that specify realistic optimizations equivalent to TVM schedules [10]. TVM is a state-of-the-art deep learning compiler [3] and [10] demonstrates that expressing optimizations performed by TVM as compositions of rewrites is possible and achieves the same high performance as TVM. The optimizations performed by TVM and Elevate are typical compiler optimizations, including loop blocking, loop permutations, vectorization, and parallelization. Overall we evaluate all seven differently optimized versions of matrix multiplication presented in [10] and described in the TVM manual.

In this evaluation, we make sure that the generated C code is equivalent modulo variable names to the manually optimized versions that already demonstrated competitive performance compared with TVM. First, we compare how much time and memory are required for fully automated equality saturation and our sketch-guided equality saturation. Then, we discuss how much programmer effort is required using manually written Elevate strategies compared to writing our sketches.

5.1 Optimization Time and Memory Consumption

Experimental Setup. Both Elevate[4] and our full Risegg implementation[5] are written in Scala and we use standard Java utilities for measurements: `System.nanoTime()` to measure runtime, and the `Runtime` api to compute maximum memory consumption by sampling regularly.

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

Fully Automated Equality Saturation. Table 2 shows the runtime and memory consumption of using a single fully automated equality saturation to perform seven different optimizations. The search terminates as soon as the optimized version is found in the e-graph. Most optimizations are not found before exhausting the 60GB of available memory. Only the "baseline" and "blocking" versions are found and the search for the blocking version requires more than 1h and about 35GB 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" include vectorization rules, while "array-packing", "cache-blocks" and "parallel" include rules for memory storage.

Sketch-Guided Equality Saturation. Table 3 shows the runtime and memory consumption of our sketch-guided equality saturation, where sketches guide the optimization process and break a single equality saturation search into multiple. All optimizations are found in less than 10s, using less than 0.5GB of RAM. Interestingly the number of rewrite rules applied by our semi-automatic approach is in the same order of magnitude as for the manual Elevate strategies [10]. 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 orchestrate the shortest rewrite path possible. The constructed e-graphs contains no more than 10^4 e-nodes and e-classes, at least two orders of magnitude less than the 10^6 required for "blocking" without sketch-guidance.

[4] https://github.com/elevate-lang/elevate

[5] https://github.com/rise-lang/shine/tree/sges/src/main/scala/rise/eqsat

Summary. While complex optimizations of matrix multiplication are not feasible with fully automated equality saturation, they are feasible with sketch-guided equality saturation. In finding all optimized version in less than 10 seconds, our semi-automated technique is practical and only about one order of magnitude slower than the Elevate strategies that perform these optimizations in under a second. Next, we investigate the programmer effort of sketch-guided equality saturation compared to orchestrating the rewrite sequence manually with Elevate.

5.2 Programmer Effort

Elevate strategies. Elevate strategies [10] are programs describing optimizations as compositions of rewrite rules. While Elevate is a functional language, strategies are inherently imperative in nature as they describe the rewrite steps required to transform the initial program into an optimized one. In contrast, sketches declaratively describe the desired programs rather than how to reach them.

Elevate enables the development of abstractions that help write concise strategies, such as the one performing the "blocking" optimization in listing 4. Unfortunately, these abstractions are often program specific and complex to implement. For example, the `reorder` abstraction in line 5 is defined as shown in listing 5. The implementation of this one abstraction is 43 lines long, involves the definition of 8 internal strategies, and carefully composes them together with more generic strategies in a recursive process. Still, this `reorder` abstraction is – despite its name – not capable of reordering generic nestings of `map` and `reduce` patterns, but only works for the matrix multiplication example. Additionally, it is hard for the programmer to reason about the list parameter which represents the desired reordering: what will the resulting program look like? Overall, the "blocking" optimization requires 112 lines of program-specific Elevate code.

The first authors of [10] and [14] that used Elevate for optimizing matrix multiplication and image processing pipelines estimated[6] that they spent between two person-weeks and one-person month to develop the Elevate strategies.

Sketches. To demonstrate their simplicity we discuss the sketches written for the matrix multiplication versions. We start by defining useful sketch abstractions that combine generic constructs from SketchBasic with Rise-specific type annotations (listing 6). In the code, `->` is a function type and `n.dt` an array type of `n` elements of type `dt`. The type annotations restrict the iteration domains of patterns like `map` and `reduceSeq`. We use similar definitions for other language constructs.

Listing 7 shows the sketch for the (basically unoptimized) "baseline" version. The sketch describes the desired program structure of two nested `map` patterns and a nested `reduce`. The comments on the right show the equivalent nested for loops.

[6] in private communication with us

[7] https://github.com/rise-lang/shine/blob/sges/src/main/scala/rise/elevate/strategies/algorithmic.scala

```
1 val blocking = ( baseline ‘;‘
2   tile(32,32)      ‘@‘ outermost(mapNest(2))   ‘;;‘
3   fissionReduceMap ‘@‘ outermost(appliedReduce)‘;;‘
4   split(4)         ‘@‘ innermost(appliedReduce)‘;;‘
5   reorder(List(1,2,5,6,3,4)))
```

Listing 4. Elevate *blocking* strategy [10]

```
1  // overall 43 lines of code
2  def reorder(l: List[Int]) =
3    normForReorder ‘;‘ (reorderRec(l) ‘@‘ topDown)
4
5  def normForReorder = // + 3 lines
6
7  def reorderRec(l: List[Int]) = e => {
8    def freduce(s)  = // + 1 line
9    def freduceX(s) = // + 1 line
10   def stepDown(s) = // + 1 line
11   def isFullyAppliedReduceSeq = // + 2 line
12   def isFullyAppliedMap       = // + 1 line
13   def moveReductionUp(pos: Int) = // + 4 lines
14       // ... moveReductionUp(pos-1) ...
15
16   l match {
17     // nothing to reorder, go further down
18     case ... => // ... reorderRec(...) ...
19     // reordering to do
20     case ... =>
21       // ... reorderRec(...) ..., + 5 lines
22     case ... => // base and failure case  + 3 lines
23   } }
```

Listing 5. Simplified version of Elevate *reorder* strategy[7]

```
1 def containsMap(n: NatPat, f: Sketch): Sketch =
2   contains(app(map :: ? -> n.? -> ?, f))
3 def containsReduceSeq(n: NatPat, f: Sketch): Sketch =
4   contains(app(reduceSeq :: ? -> ? -> n.? -> ?, f))
5 def containsAddMul: Sketch =
6   contains(app(app(+, ?), contains(×)))
```

Listing 6. Some sketch abstractions used in the evaluation

```
1 containsMap(m,          | for m:
2  containsMap(n,         |  for n:
3   containsReduceSeq(k,  |   for k:
4    containsAddMul)))    |     .. + .. * ..
```

Listing 7. Sketch for the baseline version

The sketch describing the "blocking" version in listing 8 corresponds to an optimized program where the imperative loop nests have been split and reordered such that the iteration space is chunked into blocks of $4 \times 32 \times 32$ processed by the three innermost for loops.

In contrast to the strategy in listing 4 sketches focus on the effect the optimization has on the program, rather than how the transformation is performed step-by-step. Developing sketches is significantly easier: we estimate that it took about two person-days to develop all sketches used for our evaluation, in contrast to the weeks required to develop the strategies. We also believe, that it is more intuitive to describe familiar program shapes rather than composing rules that rewrite the program.

```
containsMap(m / 32,                 | for m / 32:
 containsMap(n / 32,                |  for n / 32:
  containsReduceSeq(k / 4,          |   for k / 4:
   containsReduceSeq(4,             |    for 4:
    containsMap(32,                 |     for 32:
     containsMap(32,                |      for 32:
      containsAddMul))))))          |       .. + .. * ..
```

Listing 8. Sketch for the blocking version

```
containsMap(m / 32,                 | for m / 32:
 containsMap(32,                    |  for 32:
  containsMap(n / 32,               |   for n / 32:
   containsMap(32,                  |    for 32:
    containsReduceSeq(k / 4,        |     for k / 4:
     containsReduceSeq(4,           |      for 4:
      containsAddMul))))))          |       .. + .. * ..
```

Listing 9. Intermediate sketch specifying how to split loops.

version	sketches
blocking	split + reorder$_1$
vectorization	split + reorder$_1$ + lower$_1$
loop-perm	split + reorder$_2$ + lower$_2$
array-packing	split + reorder$_2$ + store + lower$_3$
cache-blocks	split + reorder$_2$ + store + lower$_4$
parallel	split + reorder$_2$ + store + lower$_5$

Table 4. Decomposition of each version into logical steps. A sketch is defined for each logical step.

Sketches for Guiding the Search. Using only the sketch in listing 8 enables equality saturation to find the optimized "blocking" program, but requires over 1 hour and 35GB. By guiding the search with an additional sketch shown in listing 9 we can significantly accelerate the search to less than 10s. This guiding sketch describes a program shape where the `map` and `reduce` patterns have been split but not yet reordered.

Table 4 shows how each optimization version can be described by logical optimization steps, each corresponding to a sketch describing the program after the step has been applied. Interestingly, the split sketch shown in listing 9 is a useful first guide for all optimized versions.

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 are required explaining 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 including map-fusion, 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 time increases by about 25×, still finding the optimizations in minutes but showing the importance of selecting a small set of rules.

We use a simple cost model that minimizes weighted term size. Common rules and cost models are easy to reuse.

6 Related Work

Automatic Optimization. Some compiler optimizations can be fully automated via equality saturation or heuristic searches [19, 28, 30, 36]. Although this approach can automatically yield high performance, it is not always feasible or even desirable as it lacks user control, may result in poor performance and may be too time-consuming.

Optimization Strategies. Compiler optimizations can be precisely controlled with rewriting strategies [10, 14, 31] or schedules [3, 26]. However, optimization strategies are challenging to write, non-declarative, and quickly become over-detailed and program-specific.

Equality Saturation with Bindings. We are not the first to attempt applying equality saturation to languages with binding. Willsey et al. implement a partial evaluator for the lambda calculus in [34] using explicit substitution. Although conceptually simple, this implementation has a prohibitively high performance cost as we demonstrated in section 3. In [27], Smith et al. propose *access patterns* as a way to avoid the need for binding structures when representing tensor programs. In this paper, we present practical solutions to make equality saturation with bindings more efficient.

Sketching. The idea of sketching has been introduced for program synthesis [16], along with counterexample guided inductive synthesis which combines a synthesizer with a validation procedure. Our approach in this paper is different as we target optimizations 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, we do not need a validation procedure.

Equational Reasoning with E-graphs. E-graphs were originally designed for efficient congruence closure in theorem provers [21], and are used in the Z3 theorem prover [6]. They have also recently been used for semantic code search in the Yogo tool [25]. In the realm of theorem proving, rewriting strategies can be compared to procedural proof languages [12, 24], while the sketch approach can be compared to declarative proof languages [4, 13, 22, 33].

7 Conclusion

This paper broadens the applicability of equality saturation for programming languages in two ways. We drastically improve the efficiency of equality saturation for languages with name bindings, like the lambda calculus. We propose sketch-guided equality saturation as a semi-automatic technique to scale equality saturation to complex optimizations that require long rewrite sequences. The experimental evaluation demonstrates that sketch-guided equality saturation enables seven sophisticated optimizations of matrix multiplication to be applied within seconds. The guided approach is declarative, and requires far less effort than imperative Elevate rewrite strategies. Moreover, most of the optimizations cannot be applied with fully-automated equality saturation.

Acknowledgments

We would like to thank Max Willsey for the open-source egg library and his valuable feedback; the Rise and Elevate teams for their open-source work. This work was supported by the Engineering and Physical Sciences Research Council [EP/W007940/1].

References

[1] Dan Benanav, Deepak Kapur, and Paliath Narendran. 1987. Complexity of matching problems. *Journal of symbolic computation* 3, 1-2 (1987), 203–216.

[2] Eduardo Bonelli, Delia Kesner, and Alejandro Ríos. 2000. A de Bruijn notation for higher-order rewriting. In *International Conference on Rewriting Techniques and Applications*. Springer, 62–79.

[3] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 578–594.

[4] Pierre Corbineau. 2007. A declarative language for the Coq proof assistant. In *International Workshop on Types for Proofs and Programs*. Springer, 69–84.

[5] 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.

[6] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[7] Nachum Dershowitz. 1993. A taste of rewrite systems. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*. Springer, 199–228.

[8] Maribel Fernández and Murdoch J Gabbay. 2007. Nominal rewriting. *Information and Computation* 205, 6 (2007), 917–965.

[9] Jean-Christophe Filliâtre and Sylvain Conchon. 2006. Type-safe modular hash-consing. In *Proceedings of the 2006 Workshop on ML*. 12–19.

[10] 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. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–29.

[11] 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*. ACM, 100–112.

[12] John Harrison. 1996. HOL Light: A tutorial introduction. In *International Conference on Formal Methods in Computer-Aided Design*. Springer, 265–269.

[13] Matt Kaufmann and J Strother Moore. 1996. ACL2: An industrial strength version of Nqthm. In *Proceedings of 11th Annual Conference on Computer Assurance. COMPASS'96*. IEEE, 23–34.

[14] Thomas Koehler and Michel Steuwer. 2021. Towards a Domain-Extensible Compiler: Optimizing an Image Processing Pipeline on Mobile CPUs. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 27–38.

[15] Łukasz Lachowski et al. 2018. On the complexity of the standard translation of lambda calculus into combinatory logic. *Reports on Mathematical Logic* 53 (2018), 19–42.

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

[17] Krzysztof Maziarz, Tom Ellis, Alan Lawrence, Andrew Fitzgibbon, and Simon Peyton Jones. 2021. Hashing modulo alpha-equivalence. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 960–973.

[18] Naums Mogers, Valentin Radu, Lu Li, Jack Turner, Michael O'Boyle, and Christophe Dubach. 2020. Automatic generation of specialized direct convolutions for mobile GPUs. In *Proceedings of the 13th Annual Workshop on General Purpose Processing using Graphics Processing Unit*. 41–50.

[19] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic Optimization for Image Processing Pipelines. *SIGARCH Comput. Archit. News* 43, 1 (March 2015), 429–443.

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

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

[22] Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Vol. 32. Citeseer.

[23] Pavel Panchekha, Alex Sanchez-Stern, James R Wilcox, and Zachary Tatlock. 2015. Automatically improving accuracy for floating point expressions. *ACM SIGPLAN Notices* 50, 6 (2015), 1–11.

[24] Lawrence C Paulson. 1994. *Isabelle: A generic theorem prover*. Vol. 828. Springer Science & Business Media.

[25] Varot Premtoon, James Koppel, and Armando Solar-Lezama. 2020. Semantic code search via equational reasoning. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1066–1082.

[26] 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.

[27] Gus Henry Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Luis Ceze, and Zachary Tatlock. 2021. Pure tensor program rewriting via access patterns (representation pearl). *arXiv preprint arXiv:2105.09377* (2021).

[28] 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*. ACM, 205–217.

[29] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: a functional data-parallel IR for high-performance GPU code generation. In *CGO*. ACM, 74–85.

[30] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 264–276.

[31] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. 1998. Building program optimizers with rewriting strategies. *ACM Sigplan Notices* 34, 1 (1998), 13–26.

[32] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. 2020. SPORES: sum-product optimization via relational equality saturation for large scale linear algebra. *arXiv preprint arXiv:2002.07951* (2020).

[33] Markus Wenzel and Freek Wiedijk. 2002. A comparison of Mizar and Isar. *Journal of Automated Reasoning* 29, 3 (2002), 389–411.

[34] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–29.

[35] Chenming Wu, Haisen Zhao, Chandrakana Nandi, Jeffrey I Lipton, Zachary Tatlock, and Adriana Schulz. 2019. Carpentry compiler. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 1–14.

[36] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality saturation for tensor graph superoptimization. *Proceedings of Machine Learning and Systems* 3 (2021).