Equality Saturation Theory Exploration à la Carte

ANJALI PAL, University of Washington, USA
BRETT SAIKI, University of Washington, USA
RYAN TJOA*, University of Washington, USA
CYNTHIA RICHEY*, University of Washington, USA
AMY ZHU, University of Washington, USA
OLIVER FLATT, University of Washington, USA
MAX WILLSEY, University of Washington, USA
ZACHARY TATLOCK, University of Washington, USA
CHANDRAKANA NANDI, Certora, USA

Rewrite rules are critical in equality saturation, an increasingly popular technique in optimizing compilers, synthesizers, and verifiers. Unfortunately, developing high-quality rulesets is difficult and error-prone. Recent work on automatically inferring rewrite rules does not scale to large terms or grammars, and existing rule inference tools are monolithic and opaque. Equality saturation users therefore struggle to guide inference and incrementally construct rulesets. As a result, most users still manually develop and maintain rulesets.

This paper proposes Enumo, a new domain-specific language for *programmable theory exploration*. Enumo provides a small set of core operators that enable users to strategically guide rule inference and incrementally build rulesets. Short Enumo programs easily replicate results from state-of-the-art tools, but Enumo programs can also scale to infer deeper rules from larger grammars than prior approaches. Its composable operators even facilitate developing new strategies for ruleset inference. We introduce a new *fast-forwarding* strategy that does not require evaluating terms in the target language, and can thus support domains that were out of scope for prior work.

We evaluate Enumo and fast-forwarding across a variety of domains. Compared to state-of-the-art techniques, Enumo can synthesize better rulesets over a diverse set of domains, in some cases matching the effects of manually-developed rulesets in systems driven by equality saturation.

CCS Concepts: • **Theory of computation → Equational logic and rewriting**; • **Software and its engineering → Domain specific languages**.

Additional Key Words and Phrases: Rewrite rules, equality saturation, program synthesis

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

*Both authors contributed equally to this work.

Authors' addresses: Anjali Pal, anjalip@cs.washington.edu, University of Washington, USA; Brett Saiki, bsaiki@cs.washington.edu, University of Washington, USA; Ryan Tjoa, rtjoa@cs.washington.edu, University of Washington, USA; Cynthia Richey, gannet@cs.washington.edu, University of Washington, USA; Amy Zhu, amyzhu@cs.washington.edu, University of Washington, , USA; Oliver Flatt, oflatt@cs.washington.edu, University of Washington, USA; Max Willsey, mwillsey@cs.washington.edu, University of Washington, USA; Zachary Tatlock, ztatlock@cs.washington.edu, University of Washington, USA; Chandrakana Nandi, chandra@certora.com, Certora, USA.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2023 Copyright held by the owner/author(s).
2475-1421/2023/10-ART258
https://doi.org/10.1145/3622834

1 INTRODUCTION

Equational theories in the form of rewrites ($\ell \rightsquigarrow r$) have long been used in term rewriting systems. Equality saturation engines in particular, which have seen a recent resurgence, leverage these theories to power systems in a wide variety of domains including program synthesis [Cao et al. 2023; McClurg et al. 2021; Nandi et al. 2020; Panchekha et al. 2015; Wang et al. 2020], formal verification [Coq 2022; Coward et al. 2022, 2023; De Moura and Bjørner 2008; Grannan et al. 2022; Nötzli et al. 2019], and optimizing compilers [Fu et al. 2023; Joshi et al. 2002; Koehler et al. 2021; Singh 2022; Tate et al. 2009; Wang et al. 2022; Yang et al. 2021]. A key challenge in building these systems is writing the rewrites themselves: too few rewrites can lead to missed optimizations; too many can complicate implementation and maintenance. Further, even one incorrect rewrite can compromise the soundness of the entire system.

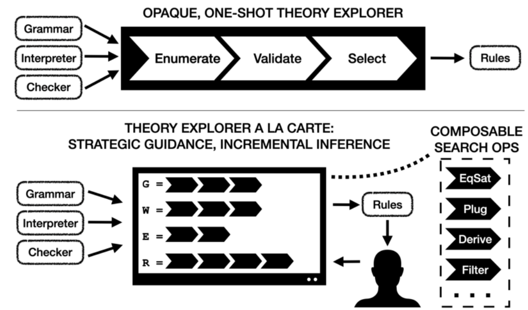


Fig. 1. *(Top)* Typical theory explorer workflow: the user provides a grammar, interpreter, and rule validator and gets a ruleset. Such theory explorers are rigid and opaque; they provide no mechanism for users to intervene or apply domain expertise to guide inference. *(Bottom)* In contrast, the Enumo DSL lets the user guide theory exploration. Users provide the same three inputs as well as a short Enumo program. Enumo's modular and composable operators make it easy to implement existing inference strategies, add domain-specific tweaks, or even implement new strategies.

Theory explorers automatically generate equational theories [Claessen et al. 2013, 2010; Johansson et al. 2010, 2014; Nandi et al. 2021; Nötzli et al. 2019; Singher and Itzhaky 2021]. These tools generally follow a three-stage approach:

(1) Enumerate terms from a given grammar, typically in a bottom-up, exhaustive manner [Barrett et al. 2011; Nandi et al. 2021].
(2) Generate candidate rewrite rules from the enumerated terms. Naively, any pair of enumerated terms could be a candidate rewrite rule. Prior work used techniques like finger-printing, fuzzing, and symbolic execution to identify "likely sound" candidates [Bansal and Aiken 2006; Nandi et al. 2021; Nötzli et al. 2019; Singher and Itzhaky 2021].
(3) Using the candidates, select a set of rewrite rules that are both sound and useful. Typically, this is done via a process that verifies the candidates and removes redundant ones. Nandi et al. [2021] call this process "minimization," under the assumption that a smaller set of rules is more likely to be effective.

Despite recent innovations, theory explorers are still not widely used. We posit that their monolithic implementations make them too inflexible. These tools are designed for idealized "one-shot" use cases: the user provides a grammar, interpreter, and verifier, presses a button, and a ruleset (set of rewrites) is produced, ready for use in a rewriting or equality saturation based system. In reality, tools based on equational theories are not developed or maintained in this manner. Instead, engineers and domain experts build, maintain, measure, debug, and compare rulesets both *iteratively* over time and *incrementally* as new features and requirements are added. In addition, automated theory explorers are often intended to replace or augment existing (handwritten) rulesets, but their rigid, one-shot approach leaves developers with little recourse when the output is not 100% satisfactory. Further, existing theory explorers do not scale to the needs of real systems. For example, the rule $x + y \rightsquigarrow \frac{x^2 - y^2}{x - y}$ is useful for factoring in numerical applications. However, discovering it via exhaustive enumeration, as shown in recent approaches [Nandi et al. 2021; Nötzli et al. 2019], is

infeasible even for a moderately sized grammar. Instead, users should be able to guide the theory explorer to discover such rules.

We present a new paradigm: theory exploration *à la carte*, which breaks theory explorers down into a set of modular operators. Users can programmatically compose these operators to easily build a theory explorer suited to their needs. To this end, we developed Enumo, an embedded domain-specific language (DSL) in which term enumeration strategies and rulesets are first-class values. Simple Enumo programs can generate useful rulesets that prior work [Nandi et al. 2021] cannot. Enumo's abstractions also inspired "fast-forwarding," a new theory exploration algorithm that supports domains where equality is undecidable (e.g., real arithmetic).

We demonstrate that Enumo programs can synthesize better rulesets compared to state-of-the-art tools, while also scaling to much larger grammars. In a case study inspired by Halide's large grammar [Ragan-Kelley et al. 2013], an Enumo program synthesized a ruleset that derives 90% of Halide's handwritten rules. Compared to prior work in theory exploration, fast-forwarding enabled Herbie [Panchekha et al. 2015] to achieve 128% higher accuracy when improving floating-point rounding error and helped us find alternate implementations of trigonometric functions in Megalibm, another floating point synthesis tool [Briggs and Panchekha 2022]. Finally, in the domain of constructive solid geometry, Enumo's synthesized rules for CAD identities let Szalinski [Nandi et al. 2020] shrink benchmarks by 87% on average, closely matching the 90% reduction achieved by expert-written rules.

In summary, this paper makes the following contributions:

- A DSL, Enumo, that offers operators for generating custom workloads, composing theory exploration strategies, and manipulating rulesets (Section 4).
- A new algorithm for "fast-forwarding" rules to infer rulesets in domains where providing an interpreter is infeasible (Section 5).
- An extensive evaluation showing that, compared to a state-of-the-art theory explorer, custom workloads and ruleset composition leads to better rulesets (Section 6).
- A set of end-to-end case studies demonstrating that Enumo's synthesized rulesets are comparable to handwritten rulesets across a variety of domains (Section 6).

2 BACKGROUND ON EQUALITY SATURATION

This paper investigates strategic theory exploration powered by the equality saturation technique and the e-graph data structure. Here, we provide a brief background on both topics.

2.1 E-graphs

An e-graph [Kozen 1977; Nelson 1980] is a data structure that efficiently represents an equivalence relation over terms, consisting of a set of *e-classes*. Each e-class is a set of equivalent *e-nodes*. An e-node $f(c_1, c_2, ...)$ is function symbol f with children e-classes c_i. An e-graph is said to *represent* a term t if any of its e-classes represents t; an e-class represents t if any e-node in the e-class represents it. An e-node $f(c_1, \ldots, c_n)$ represents a term $f(t_1, \ldots, t_n)$ if each c_i represents t_i. Two terms represented by the same e-class are considered equivalent. We additionally define some operators over e-graphs that are useful later in the paper.

- The $\text{add}(t)$ operator adds term t to the e-graph and returns the e-class that represents t.
- The $\text{lookup}(t)$ operator returns the e-class that represents a term t if such an e-class exists.
- The $\text{merge}(c_1, c_2)$ operator combines two e-class ids into a single e-class.

Willsey et al. [2021] gives the semantics of these operators in more detail. E-graphs were developed for automated theorem proving and are today used in SMT solvers [Barrett et al. 2011; De Moura and

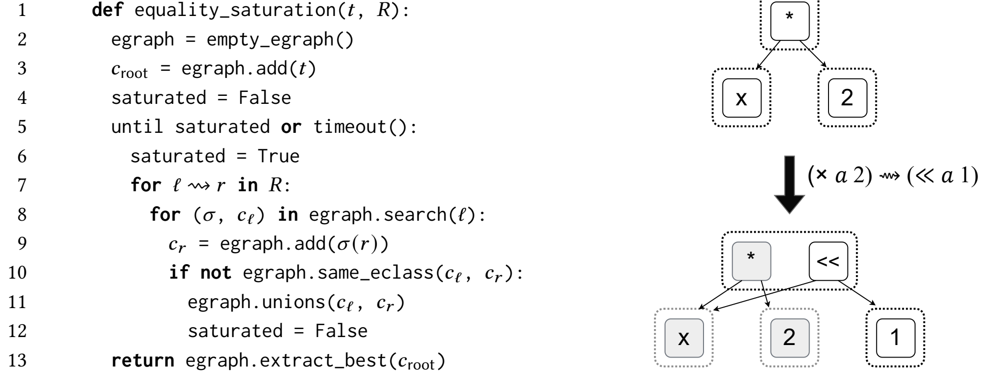


Fig. 2. (*Left*) The Equality Saturation algorithm [Nandi et al. 2021; Tate et al. 2009; Willsey et al. 2021]. Initially, a new e-graph is created that represents the input term t. Sound rewrite rules are applied until saturation or some resource bound (like iteration limit or timeout) is reached. A cost function is used to extract the "best" program from the e-graph. (*Right*) Examples of an e-graph before and after applying the rewrite $(\times\ a\ 2) \rightsquigarrow (\ll\ a\ 1)$. The dotted boxes represent e-classes, and the solid boxes represent e-nodes. The e-graph on top represents the AST (abstract syntax tree) of the input term, $(\times\ x\ 2)$, and the e-graph below shows that the e-classes that represent the terms $(\times\ x\ 2)$ and $(\ll\ x\ 1)$ are merged after the rule is applied.

Bjørner 2008]. More recently, e-graphs have been used to power a program optimization technique called equality saturation [Tate et al. 2009; Willsey et al. 2021].

2.2 Equality Saturation

Consider a term, t, and a set of rewrite rules, R. A rewrite rule, $\ell \rightsquigarrow r$, is a pair of patterns. A traditional term rewriting system applies a rewrite by finding substitutions, σ, such that $\sigma(\ell)$ is a subterm of t and then replacing the subterm with $\sigma(r)$. In this paradigm, the final output term can vary greatly depending on the order in which rewrites are applied.

Equality saturation [Tate et al. 2009; Willsey et al. 2021] is an alternative to conventional term rewriting that uses an e-graph to *non-destructively* apply rewrites. In equality saturation, matched subterms are not replaced; instead, they are merged into the same e-class.

The core algorithm is shown in Figure 2, alongside an example of an e-graph before and after a rewrite rule is applied. The equality saturation algorithm takes as input a term, t, a set of rewrite rules, R, and some resource limits (e.g., timeout, iteration limit, e-graph size in terms of number of e-nodes), and it outputs a term, t', that is equivalent to t. First, it creates a new e-graph representing t (line 3). Equality saturation then applies each rewrite rule in R to the e-graph. Rule application occurs in three stages:

(1) Line 8: an algorithm called *e-matching* [de Moura and Bjørner 2007; Detlefs et al. 2005] finds all terms in the e-graph that match the pattern ℓ. E-matching returns a list of tuples (σ, ec_ℓ), where σ is the substitution and ec_ℓ is an e-class that represents the term $\sigma(\ell)$.
(2) Line 9: for each tuple (σ, ec_ℓ), equality saturation applies σ to r to get a term $\sigma(r)$. If $\sigma(r)$ is not already represented by the e-graph, it is added in a new e-class.
(3) Line 11: the e-class representing $\sigma(\ell)$ and the e-class representing $\sigma(r)$ are merged.

Ideally, rewrites are applied until the e-graph saturates, i.e., no new e-nodes are added to the e-graph. In practice, saturation is rare, and resource bounds like iteration or e-node limits are

necessary to control termination. Following rule application, a cost function is used to extract the "best" expression equivalent to t from the e-graph

Many tools use equality saturation to drive program transformations, program synthesis, and equivalence checking [Nandi et al. 2020; Panchekha et al. 2015; Premtoon et al. 2020; Tate et al. 2009; VanHattum et al. 2021; Wang et al. 2020]. Recently, equality saturation has been used to find rulesets for other equality saturation-based systems [Nandi et al. 2021; Singher and Itzhaky 2021].

2.3 Equational Theory Inference

Theory exploration tools infer a set of axioms for a given domain. In this paper, we focus only on equational axioms, which most of these tools emit.

As described in Section 1, equational theory inference typically follows a three step process: (1) term enumeration, (2) candidate generation, (3) rule filtering. In current theory exploration tools, these steps are embedded in the core synthesis algorithm, making it difficult or impossible for users to guide or customize the tools according to their use case. This paper presents the Enumo DSL, which makes theory exploration modular and hence user-customizable by offering a small set of composable operators.

3 ENUMO BY EXAMPLE

Enumo is a DSL that provides the operators needed to build a theory explorer driven by equality saturation, for equality saturation, *à la carte*. Users define their own term enumeration strategies and can customize how the resulting rules are processed and combined.

To introduce the basics of Enumo, we walk through a simple example of learning rules over the domain of rational arithmetic. Section 3.1 recreates prior work on theory exploration ([Nandi et al. 2021]) in a few lines of Enumo code. Section 3.2 shows how enumo surpasses the capabilities of existing tools by employing workload construction operators to guide term enumeration. Section 3.3 demonstrates Enumo's ruleset manipulation primitives, including a new algorithm for learning rules without an interpreter.

3.1 Enumo Basics: Learning Rules for Rational Arithmetic

Consider the task of learning rules for the domain of rational arithmetic (a grammar with operators `+, -, ×, /, abs, ~`), where we assume the user can provide a concrete evaluator (a standard recursive interpreter). As with any theory exploration tool, the first step is to enumerate terms from the domain. This is typically accomplished by giving a grammar to the theory explorer, which then exhaustively enumerates terms up to some depth. In Enumo, however, the user constructs a *workload* that enumerates terms by composing various workload primitives and combinators. The simplest workload is a set of s-expressions:

```
1     leaves = { a b c -1 0 1 }
2     grammar = {
3       EXPR
4       (~ EXPR)
5       (abs EXPR)
6       (+ EXPR EXPR)
7       (- EXPR EXPR)
8       (* EXPR EXPR)
9       (/ EXPR EXPR)
10    }
```

The `leaves` workload is a set of six atomic s-expressions that represent the atoms of the domain: symbols `a`, `b`, `c` and constants `-1`, `0`, and `1`. The `grammar` workload is a set of s-expressions that

represent the grammar of the domain. The EXPR symbol is intended as a placeholder but has no special meaning to Enumo. By convention, we fully capitalize these placeholder symbols to differentiate them from symbols and operators in the domain language.

The plug operator lets the user compose two workloads, $\mathcal{W}_1$ and $\mathcal{W}_2$, by replacing occurrences of a given symbol x in $\mathcal{W}_1$ with s-expressions from $\mathcal{W}_2$. Using plug, the user can construct a workload that exhaustively enumerates terms up to a particular depth:

```
11    rationals_depth1 = grammar.plug("EXPR", leaves)
12    rationals_depth2 = grammar.plug("EXPR", rationals_depth1)
```

Note that $\mathcal{W}_1$.plug("x", $\mathcal{W}_2$) yields all possible combinations of replacing symbol "x" in $\mathcal{W}_1$ with an s-expression from $\mathcal{W}_2$ (Section 4, for detailed semantics). Because the grammar on line 3 includes EXPR, grammar.plug("EXPR", W) includes all s-expressions in W. Lines 11 and 12 enumerate all terms up to depth 1 and 2, respectively.

With a workload in hand that represents enumerated terms from the domain, we can now write an Enumo program to learn rules. Initially, we learn rules following the conventional approach in [Nandi et al. 2021] and [Nötzli et al. 2019]; in later sections, we show a more advanced Enumo program. First, we learn rules of depth 1 (D1):

```
13    candidates1 = rationals_depth1
14                  .to_egraph()
15                  .find_candidates()
16    (valid1, _) = candidates1.partition(|c| c.is_valid())
17    rules1 = valid1.minimize([])
```

Line 14 converts the D1 workload to an e-graph by evaluating the workload (according to the semantics in Section 4) and adding the resulting s-expressions to the e-graph. Unlike the workload's untyped s-expressions, the e-graph is typed according to the domain L (in this case, rational arithmetic). [1] Once the e-graph is constructed, we use find_candidates to generate rule candidates (line 15). This method uses the *characteristic vector* (cvec) matching approach from Nandi et al. [2021], evaluating e-graph terms with the user-provided interpreter on a sampling of constants. Terms that evaluate identically on all inputs are likely to be equivalent and are thus candidates for rules. Candidates are not necessarily sound, so we must validate them on line 16 with a user-provided verifier (over the rationals, we use Z3 [De Moura and Bjørner 2008]).

Finally, on line 17, we minimize the candidates to eliminate redundant rules. Enumo's minimize operator is parameterized over a scheduling Strategy. Using a Strategy, a client can control how much redundancy is permissible in the ruleset, measured via "derivability," as defined in Section 4.3. minimize works as follows: to get rules1, rules from valid1 are added one-by-one to a new, initially empty ruleset. A given rule $r \in$ valid1 is added to rules1 if the rules currently in rules1 *cannot* derive r within a given number of equality saturation iterations.

At this point, this small Enumo program has emulated the behavior of Nandi et al. [2021]'s Ruler tool to learn D1 rules. We can now learn rules of depth 2 (D2) by repeating a similar process:

```
18    candidates2 = rationals_depth2 # from line 12
19                  .to_egraph()
20                  .compress(rules1)
21                  .find_candidates()
22    (valid2, _) = candidates2.partition(|c| c.is_valid())
23    rules2 = valid2.minimize(rules1)
24    all_rules = rules2.union(rules1)
```

[1] An exception occurs if s-expressions cannot be parsed into the syntax for domain L.

Again following Nandi et al. [2021], learning the D2 rules benefits from the D1 rules in two ways. First, on line 20, we `compress` (see Figure 4) the D2 e-graph prior to finding candidates. This not only shrinks the e-graph and makes finding candidates more efficient but also prevents learning candidates that are implied by the D1 rules. Similarly, we pass the D1 rules to the `minimize` operator to minimize candidates not only with respect to each other but also with respect to the D1 rules. To produce the final ruleset, we compose `rules2` and `rules1` into `all_rules`.

3.2 Guided Enumeration to Find "Deeper" Rules

Notice that we just implemented the entire Ruler [Nandi et al. 2021] tool in about twenty lines of Enumo. Enumo workload operators can easily express exhaustive term enumeration, and the candidate generation and selection techniques used in Ruler are also supported in Enumo. However, while tools like Ruler can perform only a few iterations before getting stuck due to the exponential growth of enumerated terms, Enumo programs can express subsets of the term space, making it easy to scale beyond what is possible with exhaustive enumeration.

In the previous example (Section 3.1), we learned rules over the rational domain, which is problematized by rewrites involving division (/) that are only conditionally true, e.g., the rule `(/ a a)` ⇝ 1 holds only when `a` is nonzero. To address this problem, we implemented a version of the rational domain in Enumo that supports conditional expressions.

Suppose we want to find the rule `(/ a a)` ⇝ `(if a 1 (/ a a))`, which is a version of the rule `(/ a a)` ⇝ `1` that finds the equivalence between `(/ a a)` and `1` only when `a` is nonzero. The right-hand side of this rule is a large term: in the domain of rational arithmetic (`+, -, *, /`) augmented with `if`, Ruler enumerates it only after it enumerates the 3,236,142 smaller terms first. Ruler has no simple mechanism by which a domain expert can narrow the search space in order to learn deeper rules; however, Enumo enables users to leverage their domain expertise to find better, deeper rules than is possible with exhaustive term enumeration.

Our goal here is to learn conditional versions of unsound rules, so we first find the unsound rule candidates from Section 3.1:

```
1  all_candidates = candidates1.union(candidates2) # rule candidates over terms up to depth 2
2  # partition rule candidates using domain-provided rule validator
3  (sound, unsound) = all_candidates.partition(|rule| rule.is_valid())
```

Next, we construct a workload from the ruleset that checks for division by zero:

```
4  guard_wkld = { }
5  guard_pattern = { (if GUARD THEN ELSE ) }
6  for rule in unsound:
7    # domain-specific function that returns a workload consisting of terms that
8    # appear as the second argument to division
9    denominators = rule.denominators()
10   # construct terms that match the unsound rule candidate, but with a check
11   # for division by zero
12   guard_wkld.add(
13     guard_pattern.plug("GUARD", denominators).plug("THEN", rule.rhs).plug("ELSE", rule.lhs)
14   )
```

Above, `guard_pattern` (line 5) is a workload consisting of a single s-expression, which serves as a pattern for the workload we are constructing. Note that because Enumo is an embedded DSL, it is possible to encode domain-specific extensions simply by writing custom functions. For example, on Line 9, we use a domain-specific function to construct a workload consisting of terms that appear in the denominator in unsound rules. We loop over the unsound rules, incrementally building the

workload with the plug operator to make a guarded version of each unsound rule. Finally, we are ready to learn rules:

```
16    candidates = guard_wkld.to_egraph().compress(rules2).find_candidates()
17    (sound_candidates, _) = candidates.partition(|rule| rule.is_valid())
18    guard_rules = sound_candidates.minimize(rules2)
```

This step closely mirrors the process of learning D2 rules in the previous subsection: we convert the workload to an e-graph, compress the e-graph using the rules we already learned, and find candidates. Finally, we minimize the candidates, again using the existing rules. The final ruleset contains the rules `(/ a a) ⇝ (if a 1 (/ a a))` and `(/ 0 a) ⇝ (if a 0 (/ 0 a))`, both of which are useful, sound rewrite rules that avoid unsoundness when the denominator could be zero. Importantly, we found these rules without enumerating all depth-3 terms.

3.3 Learning Refined Rules with Ruleset Manipulation

We now consider an alternate approach to candidate generation. Suppose we want to learn rules for transcendental functions (e.g., trigonometric operators). For the examples in Section 3.1 and Section 3.2, we used cvec matching, which requires the user to implement an interpreter for the domain. For transcendental functions, equality is undecidable [Boehm 2020], so cvec matching is not possible. However, these functions *can* be represented in terms of other functions over rational and complex domains, for which there exist various identities. For example, the functions sine and cosine can be represented mathematically by $\sin(x) = \frac{\mathrm{cis}(x)-\mathrm{cis}(-x)}{2i}$ and $\cos(x) = \frac{\mathrm{cis}(x)+\mathrm{cis}(-x)}{2}$, where $\mathrm{cis}(x)$ is the complex exponential e^{ix} [2].

Leveraging the compositional nature of Enumo operators, we can first synthesize rewrite rules over rationals and then use them to learn rules for trigonometric functions without needing to evaluate trigonometric terms directly. We begin with a set of rewrite rules over rationals that we synthesized previously using an Enumo program.

```
1    initial_rules = Ruleset.from_file("initial.rules")
```

Next, we add *exploratory* rules (Section 5) that express the trigonometric operators in terms of the rational and complex operators. These rules are typically handwritten:

```
2    explore_rules = [
3      "(sin a) ==> (/ (- (cis a) (cis (~ a))) (* 2 I))",
4      "(cos a) ==> (/ (+ (cis a) (cis (~ a))) 2)",
5      "(tan a) ==> (* I (/ (- (cis (~ a)) (cis a)) (+ (cis (~ a)) (cis a))))"]
```

Now, we construct a workload representing trigonometric terms:

```
6    consts = { 0 (/ PI 6) (/ PI 4) (/ PI 3) (/ PI 2) PI (* PI 2) }
7    wkld = { (OP VAL) }.plug("OP", { sin cos tan }).plug("VAL", consts)
8            .filter(Filter.Not(Filter.Contains("(tan (/ PI 2))")))
```

This workload represents terms that have a single trigonometric operator applied to a constant value; notice that we can easily filter out `(tan (/ PI 2))`, which is undefined. Finally, we convert the workload to an e-graph and run the fast-forwarding algorithm to discover new rules using our prior rules:

```
9    trig_rules =
10     wkld
11       .to_egraph()
12       .fast_forward(initial_rules, explore_rules, limits)
13       .minimize(initial_rules)
```

[2] https://en.wikipedia.org/wiki/Cis_(mathematics)

$\langle workload \rangle$ ::= Set $\langle s\text{-}exp \rangle^*$
| Union $\langle workload \rangle^*$
| Filter $\langle filter \rangle$ $\langle workload \rangle$
| Plug $\langle workload \rangle$ $\langle string \rangle$ $\langle workload \rangle$

$\langle s\text{-}exp \rangle$ ::= Atom $\langle string \rangle$
| List $\langle s\text{-}exp \rangle$+

$\langle filter \rangle$ ::= MetricLt $\langle metric \rangle$ $\langle \mathbb{N} \rangle$
| MetricEq $\langle metric \rangle$ $\langle \mathbb{N} \rangle$
| Contains $\langle pattern \rangle$
| Excludes $\langle pattern \rangle$
| Canon $\langle string \rangle$+
| And $\langle filter \rangle$+ | Or $\langle filter \rangle$+ | Not $\langle filter \rangle$

$\langle metric \rangle$ ::= Atoms | List | Depth

(a) Enumo workload abstract syntax.

$$[\![\text{Set } ts]\!] = ts \qquad [\![\text{Filter } filter\ \mathcal{W}]\!] = \{t \in [\![\mathcal{W}]\!] \mid [\![filter]\!](t) = true\}$$

$$[\![\text{Union } \mathcal{W}_1\ \mathcal{W}_2]\!] = [\![\mathcal{W}_1]\!] \cup [\![\mathcal{W}_2]\!] \qquad [\![\text{Plug } \mathcal{W}_1\ tgt\ \mathcal{W}_2]\!] = \bigcup_{e \in [\![\mathcal{W}_1]\!]} [\![\text{plug_sexp } e\ tgt\ \mathcal{W}_2]\!]$$

$$[\![\text{plug_sexp } (Atom\ s)\ s\ \mathcal{W}]\!] = [\![\mathcal{W}]\!]$$
$$[\![\text{plug_sexp } (Atom\ s)\ tgt\ \mathcal{W}]\!] = \{Atom\ s\} \text{ when } s \neq tgt$$
$$[\![\text{plug_sexp } (List\ s_1\ \dots\ s_n)\ tgt\ \mathcal{W}]\!] = \{List\ t_1\ \dots\ t_n \mid t_i \in [\![\text{plug_sexp } s_i\ tgt\ \mathcal{W}]\!]\}$$

(b) Enumo workload semantics.

$$[\![\text{MetricLt } M\ n]\!](t) = [\![M]\!](t) < n \qquad [\![\text{Contains } p]\!](t) = \exists\sigma, \sigma(p) \in \text{subterms}(t)$$
$$[\![\text{MetricEq } M\ n]\!](t) = [\![M]\!](t) = n \qquad [\![\text{Canon } \vec{a}\ ]\!](t) = canon(\vec{a}, t) == t$$

(c) Enumo filter semantics. Metric filters test various measures of term size (number of atoms, number of lists, or depth), e.g., (+ a b) has 3 atoms, 1 list, and depth 2. The Contains filter tests whether any subterm of t matches a pattern p. The Canon filter tests whether t is *canonical* with respect to a vector of atoms $\vec{a}$, e.g., Canon $[a, b, c]$ (+ a b c) is true while Canon $[a, b, c]$ (+ b a c) is false. And, Or, and Not have the standard semantics. Exludes is simply the negation of Contains.

Fig. 3. Syntax and semantics for the workload fragment of the Enumo DSL.

We explain the fast-forwarding algorithm in detail in Section 5, but at a high level, it identifies candidates by running known rewrite rules and considering merged e-classes as rule candidates. By definition, all rules found using this algorithm are derivable from the starting ruleset, but the rulesets generated can still be valuable in practice. In this case, fast-forwarding lets us find rules over the trigonometric operators directly, rather than needing to rewrite through large terms with complex operators.

4 ENUMO: A DSL FOR STRATEGIC THEORY EXPLORATION

This section presents the core of the Enumo DSL for guided term enumeration and incremental rewrite rule inference. Enumo programs primarily manipulate two kinds of values: *workloads*, which represent sets of terms, and *rulesets*, which are sets of pairs of patterns. Both terms within workloads and patterns within rewrite rules are represented as (untyped) s-expressions.

Enumo programs typically iterate the following steps:

(1) Construct a workload $\mathcal{W}$ representing a search space with terms of interest.
(2) Convert $\mathcal{W}$ to an e-graph and use the current ruleset to merge equivalent e-classes.
(3) Search this compressed e-graph to find candidate rewrite rules, i.e., unmerged pairs of e-classes that fuzzing or other techniques suggest may be equivalent.

(4) Minimize the set of candidates by removing rules that are unsound or redundant given the current ruleset.
(5) Add the set of minimized candidates to the current ruleset.

Enumo provides several operators for constructing and manipulating both workloads and rulesets, including *plugging* and *iterating* workloads to build up sets of terms, *forcing* to materialize and insert a workload of terms into an e-graph, *searching* e-graphs built from workloads for candidate rewrite rules, and *minimizing* rulesets to remove redundant rules. Enumo programs are essentially a sequence of bindings from variables to workload and ruleset expressions embedded in a host programming language, e.g., a simple lambda calculus.

4.1 Workloads

Figure 3 shows the syntax and semantics of workloads in Enumo. Workloads have four constructors: (1) Set represents a literal set of s-expressions, (2) Union represents unions of workloads, (3) Filter represents a subset of terms in a workload, and (4) Plug represents substituting one workload into another. Note that workloads **represent** sets of terms, but do not eagerly materialize them. In Enumo, workloads are typically materialized into an e-graph using the to_egraph operator. This laziness is a key design decision that lets Enumo programs efficiently represent and manipulate large sets of terms.

Set and Union have straightforward semantics (Figure 3b). Filter takes a workload $\mathcal{W}$ and a filter predicate P, and represents the set of terms from $\mathcal{W}$ that satisfy P. Figure 3a shows the syntax of filter predicates. Some filters use term *metrics*, which count the number of atoms, lists, and depth of a term. The semantics for filters is given in Figure 3c. The MetricEq and MetricLt filters measure a metric of a term and compare it to a given value; the Contains filter checks whether the given pattern occurs in a term; the Excludes filter is the inverse of Contains; the Canon filter checks that a given term is canonical with respect to a given list of variables; and the Not, Or, and And filters are the usual logical connectives.

Plug lets the user substitute all combinations of terms from one workload for a given variable in another workload. As the semantics in Figure 3b show, Plug provides a special kind of substitution that performs a Cartesian product: Plug $\mathcal{W}_1$ s $\mathcal{W}_2$ returns a workload that denotes a set of $\sum_{i=0}^{|\mathcal{W}_1|} |\mathcal{W}_2|^{m_i}$ terms, where there are m_i occurrences of s in each term, t_i, in $\mathcal{W}_1$. The following Enumo snippet demonstrates the semantics of Plug:

```
1  w1 = { X (foo X X) }
2  w2 = { 1 y }
3  plugged = w1.plug("X", w2) # plugged == { 1 y (foo 1 1) (foo 1 y) (foo y 1) (foo y y)}
```

Enumo's operators can be composed into useful, reusable strategies beyond the concise reimplementation of past work. As an example, iter_metric, defined in the following Enumo snippet, can be used to create size-parameterized workloads: iter_metric(W, tgt, Atoms, n) produces all terms from the workload with at most n atoms, and iter_metric(W, tgt, Depth, n) produces all terms from the workload up to depth n. iter_metric can be used to generate workloads with successively larger terms and thus guide the exploration of successively deeper rules across domains (Section 6).

```
1  def iter_plug(W, tgt, n):
2    if n <= 0: return W
3    return W.plug(tgt, iter_plug(W, tgt, n - 1))
4
5  def iter_metric(W, tgt, metric, n);
6    return iter_plug(W, tgt, n).filter(metric <= n)
```

E-GRAPH GENERATING OPERATORS
to_egraph: workload → egraph
eqsat : egraph → ruleset → egraph
compress : egraph → ruleset → egraph

RULE TESTING OPERATORS
is_saturating : rule → bool
is_valid : rule → bool
can_derive : ruleset → rule → bool

RULESET GENERATING OPERATORS
find_candidates : egraph → ruleset
partition : ruleset → (rule → bool) → ruleset * ruleset
minimize : ruleset → ruleset → ruleset
union : ruleset → ruleset → ruleset
candidates_by_diff : egraph → egraph → ruleset

Fig. 4. ENUMO's operators over rulesets and e-graphs.

Optimizing Workloads. Plug is the key workload combinator for representing search spaces by enumerating terms from a grammar. Plug typically represents combinatorially many more terms than its arguments, but the result of a Plug is often Filtered to target a more specific subset of the represented terms. We introduce an essential optimization to speed up workload evaluation that avoids unnecessary work during combinatorial substitution by pushing *monotonic* Filters through Plugs according to the following equation:

$$[\![\text{Filter } \mathit{filter}\ (\text{Plug } \mathcal{W}_1\ s\ \mathcal{W}_2)]\!] = [\![\text{Filter } \mathit{filter}\ (\text{Plug } \mathcal{W}_1\ s\ (\text{Filter } \mathit{filter}\ \mathcal{W}_2))]\!]$$

A filter f is *monotonic* if, for every term t satisfying f, every subterm $s \in t$ also satisfies f. Note that the outer Filter remains in place even after the optimization since removing it entirely would not preserve semantics. This still provides exponential speedups in the number of terms that must be filtered. *All ENUMO programs in our evaluation depend heavily on this optimization.*

In the current ENUMO implementation, And, Excludes, and MetricLt are monotonic filters, so they are pushed through Plugs. This optimization, and its monotonicity constraint, are inspired by the classic relational algebra optimization of pushing certain selections through joins [Abiteboul et al. 1995]; in some ways, Plug's combinatorial behavior resembles a relational join.

4.2 E-graphs and Rulesets

In addition to novel, programmable term enumeration, ENUMO also provides primitives to create and manipulate e-graphs and rulesets. Figure 4 shows these operators and their types. Many mirror parts of earlier monolithic theory explorers; ENUMO's key insight lies in turning such tools "inside out" to expose their components as composable operators in a DSL that lets users strategically guide the search for rewrites and incrementally build up inferred rulesets.

E-graph Operators. In the ENUMO language definition, an e-graph is an abstract data type that provides the operations described in Section 2.1. A typical ENUMO program (Section 3), converts a workload into an e-graph using the to_egraph operator before generating candidates. The resulting e-graph represents every term in the set denoted by the workload.

The `eqsat` operator runs equality saturation on the given e-graph with the given ruleset. From ENUMO's perspective, `eqsat`'s main purpose is to remove redundancy from the e-graph implied by a ruleset of already-learned rewrites.

The `compress` operator also runs equality saturation, but it does not allow the e-graph to grow. It runs equality saturation on a copy of the e-graph and backports only the unions. Section 5 shows how these strategies affect rule inference.

Ruleset Operators. A ruleset is a set of rewrite rules, where each rule is a pair of patterns. Rulesets can be read from or written to a file, manipulated using the ruleset operators in Figure 4, and used to perform equality saturation on e-graphs using the `eqsat` operator described previously.

In a typical Enumo program, the find_candidates operator is used to infer a ruleset from an e-graph. find_candidates is parameterized on a user-provided interpreter that identifies likely sound rule candidates by evaluating the terms over a set of inputs (fuzzing); terms that disagree are certainly not equivalent, but those that agree may be [Bansal and Aiken 2006; Nandi et al. 2021]. To define find_candidates formally, let $\mathrm{repr}(e)$ denote a *representative* term from e-class e, and let $\mathrm{eval}(t)$ denote the result of evaluating t over some set of input values. Then, find_candidates on e-graph E returns a set of rules:

$$\{\mathrm{repr}(e_l) \rightsquigarrow \mathrm{repr}(e_r) \mid e_l, e_r \in E.\ \mathrm{eval}(\mathrm{repr}(e_l)) = \mathrm{eval}(\mathrm{repr}(e_r))\}$$

The partition operator takes a ruleset R and a predicate P over rules and returns (R_1, R_2) such that $R = R_1 \cup R_2$, $r \in R_1 \implies P(r)$, and $r \in R_2 \implies \neg P(r)$. Figure 4 shows two such predicates, is_valid and is_saturating. The is_valid predicate checks whether a rule $\ell \rightsquigarrow r$ is valid for all inputs using a user-provided verifier for the domain. Depending on the domain, the verifier can use techniques like SMT, model checking, or fuzzing. The built-in is_saturating predicate checks whether a rule is *saturating*, i.e., applying the rule to an e-graph will not increase its size, At a high level, saturating rules have a right-hand side pattern that contains only subterms appearing in the left-hand side, except potentially for the root operator. For example, $x + y \rightsquigarrow y + x$ is saturating since all non-root subterms in the right-hand side (x and y) also occur in the left-hand side, but $x + (y + z) \rightsquigarrow (x + y) + z$ is not since the right-hand side contains a non-root subterm $x + y$ that does not appear in the left-hand side. Applying only saturating rules to an e-graph is guaranteed to reach a fixpoint past which further application of the rules no longer changes the e-graph.

The can_derive operator tests whether a ruleset R can derive a rule $\ell \rightsquigarrow r$, discussed in Section 4.3. The minimize operator takes a ruleset R and prior rules P and *minimizes* R with respect to P, guaranteeing that $\mathtt{minimize}(R, P) = R' \implies \forall r \in R \setminus R', \mathrm{can_derive}(P \cup R', r)$. Conceptually, this filters R to a small subset R' of rules such that $r \in R' \implies \neg\mathrm{can_derive}(P, r)$. However, Enumo's minimize operator provides an optimization that batches these checks to simultaneously eliminate redundant rules, initially described in Nandi et al. [2021].

The final core operator is candidates_by_diff, which takes two e-graphs e_1 and e_2 and returns a ruleset. candidates_by_diff infers candidate rewrites rules from e-classes that merged during an equality saturation run, i.e., terms that a given ruleset could prove equivalent. Typically, e_2 is the result of running equality saturation on e_1 with ruleset R. If, by application of R, the equivalence between terms t and t' is discovered, then candidates_by_diff learns a rule candidate by extracting the best expression from the e-classes representing t and t' in e_1. candidates_by_diff enables rule synthesis for new domains, which prior work could not support. This utility is briefly exemplified in Section 3.2 and formally presented in Section 5.

4.3 Discussion of Derivability

Given two rulesets, how do we know which is better? While it may be tempting to use ruleset size as a proxy for ruleset quality, more rules are not necessarily preferable because overly redundant rules degrade the performance of equality saturation systems. A small set of simple rules is often easier to maintain and debug than a large set of complicated ones. On the other hand, a ruleset with too few rules is less useful because fewer equivalences will be found, especially since resource limits restrict the number of iterations of equality saturation. Since saturation is rare in practice, it is often helpful to have *some* redundancy in the rulesets to improve results under given resource limits (see Section 5). Quantifying a ruleset's *proving power* under given resource limits is subtle and difficult to estimate. In this section, we define ruleset *derivability*, a metric for measuring proving power that we use to compare rulesets.

Derivability. Prior work has not established a standard definition of derivability in the context of equality saturation. In this paper, we formalize two "obvious" definitions of derivability: LHS-RHS and LHS. To test whether a ruleset, R, can derive a rule, $\ell \rightsquigarrow r$, under given resource limits, we use the equality saturation procedure (Figure 2). The function `timeout` determines when to stop the equality saturation loop based on available resources (e.g., node count, iteration, or time bounds). If running equality saturation using ruleset R causes e-classes representing ℓ and r to merge, we say $\ell \rightsquigarrow r$ is *derivable* from R under the given resource bounds. The LHS-RHS derivability metric measures whether the equivalence between ℓ and r can be recovered by applying the rules in R to an e-graph initialized with both ℓ and r. In contrast, the stronger LHS definition for derivability states that $\ell \rightsquigarrow r$ can be recovered given only ℓ. Prior work used the LHS-RHS definition of derivability [Nandi et al. 2021].

In the context of equality saturation, the initial state of the e-graph interacts with resource limits in subtle ways because it changes what terms are available during e-matching. Rules in R must find concrete terms in the e-graph that match the left side of the rule in order to add the right side and merge the two e-classes. Changing the initialization of the e-graph thus changes the rule matches that are possible.

To illustrate the difference between LHS and LHS-RHS derivability, consider the rule $a \rightsquigarrow b$ (where a and b are arbitrary patterns) and a ruleset containing the rule $b \rightsquigarrow a$. In an e-graph initialized with both a and b (LHS-RHS), the rule $b \rightsquigarrow a$ fires and the e-classes merge, so the rule $a \rightsquigarrow b$ is considered derivable. In an e-graph initialized with just a (LHS), $b \rightsquigarrow a$ does not fire, so the rule $a \rightsquigarrow b$ is not considered derivable. LHS and LHS-RHS derivabilities can also require different resource limits. For example, consider using $R = \{a \rightsquigarrow b, b \rightsquigarrow c, c \rightsquigarrow b\}$ to derive the rule $a \rightsquigarrow c$. Under LHS-RHS, the e-classes representing a and c merge within a single iteration of equality saturation. In contrast, using LHS derivability, recovering the equality between a and c takes two iterations of equality saturation. First, the rule $a \rightsquigarrow b$ fires, creating an e-class for b and merging it with a's e-class. In the second iteration, the rule $b \rightsquigarrow c$ fires, creating an e-class for c and merging it with the e-class that represents a and b, thus recovering the equivalence between a and c. This example shows that LHS-RHS derivability may be able to derive equivalences in fewer iterations (i.e., using less resources) than LHS because it can match on the left- and right-hand sides simultaneously.

In general, LHS derivability is more conservative. Anecdotally, we find that it is preferable when the user is interested in optimization-based equality saturation applications, where an e-graph is initialized with a single term t and equality saturation is used to find a better, equivalent version of t. In contrast, LHS-RHS derivability is looser but may be appropriate in equivalence-checking equality saturation applications, where two terms t_1 and t_2 are added to an e-graph and an equality saturation engine like egg is applied to see if their e-classes merge.

5 A FAST-FORWARDING THEORY EXPLORER

This section presents a new *fast-forwarding* algorithm for theory exploration, which has two key applications. First, as Section 3.3 showed, it enables rewrite rule inference for domains where writing an interpreter is prohibitively difficult. Second, it mitigates the effect of resource limits on the performance of rewrite-driven systems that are often caused by the kind of rules used.

The *kinds* of rules that comprise a ruleset significantly affect performance, even in efficient equality saturation-driven systems. Since reaching saturation in an e-graph is rare in practice, iteration and/or node limits are used to ensure termination. As a result, two rulesets can have vastly different performance even if they are equivalent under derivability, as shown in Section 4.3.

The key motivation behind the fast-forwarding algorithm is that the "right" set of rules can help *fast-forward* equality saturation by skipping intermediate derivations. Skipping intermediate

```
1 def fast_forward_naive(𝒲, ℛ):
2   𝒢 = 𝒲.to_egraph() # convert workload to e-graph
3   𝒢′ = 𝒢.compress(ℛ) # run equality saturation with all rules
4   candidates = candidates_by_diff(𝒢, 𝒢′) # any two terms from the unions are potential candidates
5   return candidates.minimize(ℛ) # minimize candidates
```

Fig. 5. A naive algorithm for fast-forwarding theory exploration that applies equality saturation to terms represented by $\mathcal{W}$ using compress.

derivations has two benefits: (1) it requires fewer iterations to prove a target equivalence, and (2) it often reduces the number of intermediate terms in the e-graph, which reduces unhelpful rewriting on these terms. Determining the "right" rules requires domain knowledge and depends on the application. To that end, we assume that the user can provide a set of *allowed* ($\mathcal{A}$) and *forbidden* ($\mathcal{F}$) operators. We then say that if a pattern, p, contains *any* operator, $o \in \mathcal{F}$, then p is forbidden. If *all* operators in p are allowed, then the pattern is allowed. Since a rule is simply a pair of patterns, these definitions extend to rules.

For the trigonometric rule synthesis task in Section 3.3, the allowed operators are `sin`, `cos`, `tan`, `PI`, `+`, `-`, `×`, etc., and the forbidden operators are `cis` and `I` because we wanted Enumo to synthesize rewrite rules over the trigonometric domain only and *not* contain `cis` and `I`. Recall that this task also required an additional set of *exploratory* ($\mathcal{E}$) rewrite rules that relate terms with allowed operators to terms with *other* operators. Crucially, these "other" operators can be *both* allowed or forbidden. The intuition behind $\mathcal{E}$ is that it helps *explore* new equivalences between allowed terms in the e-graph by applying a *known* set of rewrites over terms containing the *other* operators (shown by `explore` in Section 3.3).

Figure 5 shows a naive implementation of fast-forwarding using the set of core Enumo operators from Section 4. The process consists of applying eqsat to a workload representing *allowed* terms using a ruleset, $\mathcal{R}$, that contains both allowed and forbidden rules. First, the algorithm creates an e-graph from the terms obtained by evaluating the workload. Then, it *shrinks* the e-graph using the compress operator; compress is an equality saturation *strategy* (Figure 4) that prevents the e-graph from getting intractably large. The fast-forwarding algorithm then applies the rewrites on a duplicate of the original e-graph and copies only the equivalences back, adding no new e-nodes or e-classes in the original e-graph. The next step in the algorithm extracts candidates from $\mathcal{G}$ based on the equalities discovered in $\mathcal{G}'$ using a cost function that penalizes forbidden operators. Finally, it minimizes the resulting ruleset as explained in Section 4. Notice that this naive algorithm simply performs a single phase of compress with *all* the rules.

A Practical Algorithm. Unfortunately, the naive algorithm in Figure 5 does not find useful rules in practice for two reasons. First, it does not scale to large workloads, which cause the e-graph to become too large before resource limits (e.g., timeout, iteration bounds) are exhausted. Second, exploring in a breadth-first manner prevents finding interesting fast-forwarding opportunities, which only occur after several rounds of equality saturation.

Instead, we propose a more practical, approximate algorithm that applies equality saturation more strategically by leveraging a user's domain knowledge in the form of $\mathcal{E}$, $\mathcal{F}$, and $\mathcal{A}$. The algorithm in Figure 6 *selectively* grows and compresses the e-graph using rules provided by the user. It first creates an e-graph from the terms represented by $\mathcal{W}$, then compresses the e-graph with allowed rules (line 3 - line 4). This step *shrinks* the e-graph with known equivalences. `fast_forward` does not learn new rule candidates at this point, because any candidates it could learn are already derivable from `allowed`. In the next step, the algorithm *grows* the e-graph with $\mathcal{E}$ (line 5). Crucially, this step

```
1 def fast_forward(𝒲, ℛ, ℰ):
2   𝒢 = 𝒲.to_egraph()  # convert workload to egraph
3   allowed = {r ∈ ℛ | ∀ o ∈ (ops(r.lhs) ∪ ops(r.rhs)), o ∈ 𝒜}
4   𝒢′ = 𝒢.compress(allowed)  # compress the egraph with allowed rules
5   𝒢″ = 𝒢′.eqsat(ℰ)  # grow the egraph with exploratory rules
6   candidates = candidates_by_diff(𝒢′, 𝒢″) # extract learned rules with no ops in ℱ
7   𝒢‴ = 𝒢″.compress(ℛ)  # compress to find equalities with all of ℛ
8   candidates.union(candidates_by_diff(𝒢″, 𝒢‴)) # add more candidates with no ops in ℱ
9   return candidates.minimize(allowed) # minimize candidates
```

Fig. 6. A practical, fast-forwarding theory exploration algorithm that approximates the naive version. *op* is a helper function that returns all the operators in a term.

does *not* use compress; it performs simple eqsat that introduces new terms and equivalences in the e-graph. Next, we learn rule candidates using `candidates_by_diff`. The final equality saturation step applies another round of compression using all the rules in $\mathcal{R}$, discovering additional rule candidates. The minimization step in this algorithm uses the allowed rules instead of the entire ruleset to avoid forbidden operators in the minimized ruleset.

Comparing Different Scheduling Strategies

The key idea in our fast-forwarding algorithm is to perform equality saturation in phases, using subsets of $\mathcal{R}$ to selectively grow and compress the e-graph. To understand how this affects performance, we ran an experiment to evaluate and compare the difference between using `eqsat` and `compress` in Figure 6. We ran four variants of the fast-forwarding algorithm using a workload of 287 terms from the domain of trigonometric operators (*sin*, *cos*, *tan*, π, $\pi/2$, etc.). Table 1 shows results of the comparison. The first two rows, which use `eqsat` in all three phases, do not terminate within 20 minutes. These variants of fast-forwarding demonstrate the importance of `compress`, which does not allow the e-graph to grow. The next two rows use `compress` in all three phases. The third row does not split up the rules in $\mathcal{R}$ and simply runs `compress(`$\mathcal{R}$`)` three times. The fourth row compresses the allowed rules ($\mathcal{A}$) in Phase 1, the exploratory rules ($\mathcal{E}$) in Phase 2, and all rules ($\mathcal{R}$) in Phase 3. The third and fourth variants both finish within seconds, but they do not find any new rules because none of the equality saturation phases allowed the e-graph to grow. These variants demonstrate the importance of the `eqsat` operator. The approach in the last row, which corresponds to the actual fast-forwarding algorithm described in Figure 6, finds 4 useful trigonometric identities in about 3 minutes. This experiment demonstrates the importance of using `eqsat` and `compress` together to strategically grow and compress the e-graph.

Table 1. Comparing `compress` and `eqsat` with different subsets of $\mathcal{R}$ for the three phases of Figure 6. $\mathcal{A}$ is the allowed rules of $\mathcal{R}$, and $\mathcal{E}$ is the exploratory rules of $\mathcal{R}$. The last row corresponds to Figure 6, showing that it is the fastest to produce a good ruleset.

Phase 1	Phase 2	Phase 3	Time (s)	# Rules with Trig Operators
eqsat ($\mathcal{R}$)	eqsat ($\mathcal{R}$)	eqsat ($\mathcal{R}$)	Timeout	-
eqsat ($\mathcal{A}$)	eqsat ($\mathcal{E}$)	eqsat ($\mathcal{R}$)	Timeout	-
compress ($\mathcal{R}$)	compress ($\mathcal{R}$)	compress ($\mathcal{R}$)	18.66	0
compress ($\mathcal{A}$)	compress ($\mathcal{E}$)	compress ($\mathcal{R}$)	9.42	0
compress ($\mathcal{A}$)	eqsat ($\mathcal{E}$)	compress ($\mathcal{R}$)	175.78	4

6 EVALUATION AND CASE STUDIES

Implementation. Enumo is an embedded DSL, implemented as a Rust library. The entire implementation is 3095 LOC, including unit tests but excluding implementations of various domains. The domains together add another 4430 LOC, which contain a grammar, evaluator, and validator for each basic domain and a grammar and fast-forwarding rules for each domain employing fast-forwarding. The various Enumo programs sum to 718 LOC. Our implementation, together with all the domains and Enumo programs, is publicly available [3].

To evaluate the contributions of this paper, this section answers the following research questions.

(1) How does guided enumeration in Enumo compare to prior work on rewrite rule synthesis? (Section 6.1)
(2) Can Enumo scale to larger grammars than existing tools can handle? (Section 6.1)
(3) Can Enumo's fast-forwarding algorithm enable rule inference for new domains that prior work could not support? (Section 6.2)
(4) How does fast-forwarding impact client applications in terms of performance and results? (Section 6.2)
(5) Do Enumo's abstractions enable cross-domain rule synthesis technique? (Section 6.3)

6.1 Guided Search with Enumo

To evaluate Enumo's guided search, we conducted the following experiments on a 64-bit Linux machine with 32 GB RAM, running Ubuntu 22.04.2 LTS.

Table 2. Results comparing Enumo to Ruler. $R_1 \rightarrow R_2$ indicates using R_1 to derive R_2 rules. We report on both LHS and LHS-RHS derivability, separated by commas. The numbers in parentheses are times in seconds.

Domain	Enumo LOC	# Enumo (Time)	# Ruler (Time)	Enumo → Ruler (Time)	Ruler → Enumo (Time)
bool	44	64 (0.35)	51 (0.05)	100% (0.01), 100% (0.01)	87.5% (5.29), 96.9% (0.01)
bv4	21	180 (7.13)	84 (0.96)	100% (0.17), 100% (0.03)	38.3% (3.67), 41.1% (4.32)
bv32	20	120 (48.78)	78 (13.1)	100% (0.15), 100% (0.01)	58.3% (1.41), 60.0% (2.08)
rational	51	131 (59.82)	113 (97.9)	94.7% (606.25), 100% (0.09)	62.6% (35.76), 68.7% (39.91)

6.1.1 Comparing Rulesets with Prior Work. Ruler [Nandi et al. 2021] is a state-of-the-art tool for automatically synthesizing rewrite rules that targets equality saturation driven systems. It uses a one-shot approach for rule synthesis. We compare the rules generated by Enumo and Ruler, finding that rulesets from small Enumo programs outperform those from Ruler. We wrote Enumo programs for each domain showcased in Ruler: `bool`, `bv4`, `bv32`, and `rational`. These programs call recursive_rules, an Enumo-provided utility function (Figure 7). From a user-provided grammar $\mathcal{G}$ that specifies literal terms, unary operators, and binary operators, recursive_rules builds workloads of increasing size; it then finds and validates rules from the workloads, using rules it finds along the way to avoid redundancy in the final ruleset. This function replicates Ruler's core loop in just a few lines, highlighting the expressivity of Enumo.

After running our Enumo programs, we compared the derivability of its generated rulesets to those produced by Ruler using the same grammar and interpreter. For rational arithmetic, we found that Ruler learns rules over division by assuming that the denominator is not zero [4]. We removed this unsound assumption and re-synthesized rational arithmetic rules.

[3] https://github.com/uwplse/ruler/tree/oopsla23-aec
[4] To mitigate the resulting unsoundness, Ruler used a custom rule application strategy from the egg [Willsey et al. 2021] library.

```
1   lang = { LIT (UOP EXPR) (BOP EXPR EXPR) }
2   def recursive_rules(𝒢, metric, n):
3     if n == 0:
4       return []
5     rec_rules = recursive_rules(𝒢, n - 1, metric)
6     workload =
7       iter_metric(lang, "EXPR", metric, n)
8         .plug("LIT", 𝒢.lits)
9         .plug("UOP", 𝒢.uops)
10        .plug("BOP", 𝒢.bops)
11    rules_n =
12      terms
13        .to_egraph()
14        .compress(rec_rules)
15        .find_candidates()
16        .minimize(rec_rules)
17    return rec_rules.extend(rules_n)
```

Fig. 7. The recursive_rules function. $\mathcal{G}$ is a struct that specifies a grammar, containing workloads for literal expressions, unary operators, and binary operators. metric is the Enumo metric used to define an upper bound on terms, and n is the size limit. recursive_rules incrementally builds a ruleset by learning rules over terms from the provided grammar of increasing size up to the specified size limit.

We also found that Enumo rulesets could derive (Section 4.3) all of Ruler's rules using Ruler's own LHS-RHS derivability metric (Table 2). The reverse is not true. Using the more conservative LHS metric, Enumo rulesets derive a higher percentage of Ruler rulesets than the reverse. Both measures suggest that Enumo rulesets have greater proving power than their Ruler counterparts.

6.1.2 Scaling to Large Grammars: The Halide Case Study. Halide [Ragan-Kelley et al. 2013] is a programming language for high-performance image processing. A major component of the Halide compiler is a traditional term rewriting system [Newcomb et al. 2020] that performs optimizing program transformations using a set of handwritten rules. Halide has a large grammar, totalling 17 boolean, arithmetic, and comparison operators. It does not use an equality saturation engine for applying the rewrite rules; nevertheless, inspired by the domain, particularly due to the size of its grammar, we developed an Enumo program to evaluate the scalability of Enumo's workload-guided strategy.

We collected Halide's handwritten ruleset by scraping source files from a recent commit in the Halide repository [5] and removing rules we could not parse, i.e., rules with side conditions, unsupported operators, and unbound variables on the rule's right-hand side. After this process, we were left with 725 rules.

To see how prior work [Nandi et al. 2021] would perform on a large domain, we implemented the Halide grammar in Ruler. Ruler's implementation ran for just one iteration (further iterations did not terminate), synthesizing a total of 90 rules in 3 seconds. This ruleset derived only 18 of the 725 original rules (2.5%) using both derivability metrics (LHS, LHS-RHS).

Without leveraging guided search, Enumo scales similarly to Ruler. A simple Enumo program that exhaustively enumerates Halide terms up to size 5 derived 309 of Halide's 725 rules. The exhaustive Enumo program outperforms Ruler only because Ruler enumerates by depth, which

[5] https://github.com/halide/Halide/commit/e7f78600e10956b44e8f214c686f310211b0d836

grows much faster than size. An Enumo program that enumerates by depth times out after depth 2 and learns rules that can derive only 13 of Halide's rules, similar to Ruler's behavior.

However, the key benefit of Enumo's guided enumeration is decoupling the grammar and the workload size. In Ruler, terms are enumerated exhaustively from the grammar up to a certain size, so with a larger grammar, Ruler hits resource limits faster. In contrast, term enumeration in Enumo is separate from the grammar itself, letting users define workloads that represent different subsets of the search space. Enumo's operators make it is easy to compose workloads, enabling a piecewise rather than total approach to term enumeration. This composability makes it possible to synthesize rulesets that are larger and deeper than would be possible with a one-shot theory exploration tool like Ruler.

To evaluate whether Enumo's guided search could help to find deeper, more complex Halide rules, we wrote a 141-line Enumo program that leveraged both exhaustive and custom enumeration. First, we exhaustively enumerated terms over subsets of Halide's boolean, arithmetic, and comparison operators [6] —up to 5 atoms, beyond which point this strategy becomes computationally infeasible. We then enumerated terms with *all* of Halide's operators up to 4 atoms in size. Finally, we created custom workloads guided by domain knowledge, selectively generating terms too large to be found using the exhaustive approach, such as `(select a (min b c) (max d c))`. These workloads leverage Enumo features such as `canon` (Section 4), which eliminated many duplicate terms, reducing one workload from 52, 491 to 9, 233 terms—an 82% decrease.[7] Ultimately, our Enumo program produced a ruleset of 845 rules capable of deriving 80.7% and 90.6% of the handwritten ruleset using the LHS and LHS-RHS derivability metrics, respectively. The handwritten Halide rules derived just 6.5% (LHS) and 10.9% (LHS-RHS) of Enumo's 845 rules, suggesting the hypothesis that theory explorers could increase the proving power of industrial rewrite-driven optimizers.

As mentioned in Section 4.3, larger rulesets are not necessarily better than smaller rulesets. In this case study, however, the smaller ruleset (from one iteration of Ruler) has measurably less proving power than the larger one (from the Enumo program), so the additional rules are justified. Synthesizing rulesets for large grammars is not feasible with tools that rely on exhaustive term enumeration, but with Enumo, it is possible to build rulesets incrementally; therefore, grammar size is not a limiting factor. **This section shows that a small program using Enumo's novel guided search finds better rules than is possible using state-of-the-art tools.**

6.2 Fast-Forwarding

In this section, we evaluate the fast-forwarding algorithm (Section 5) in two new domains to learn rewrite rules that other state-of-the-art tools do not support. We also show that synthesized rules from Enumo can be easily integrated with existing equality saturation-based synthesis tools.

6.2.1 Numeric Domain. We used both the fast-forwarding algorithm and guided enumeration to infer rewrite rules for the domain of transcendental functions. To evaluate the quality of the rulesets, we integrated the rules into two existing rewrite-rule based synthesis tools, Herbie [Panchekha et al. 2015] and Megalibm [Briggs and Panchekha 2022].

Trigonometric and Exponential Representation. Recall that sine and cosine have representations in terms of the complex exponential $\mathrm{cis}(x)$ (Section 3.3). Using 57 automatically generated arithmetic rules and 15 handwritten rules using the complex exponential, we derived rules for sin, cos, and

[6]For both Ruler and Enumo, we enumerated terms over 15 of the 17 operators, skipping / and $\implies$ since most of those rules had side conditions.

[7]The full Enumo program can be found at https://github.com/uwplse/ruler/tree/main/tests/recipes.

Table 3. Derivability comparison between rules from Enumo and Herbie. As in Table 2, $R_1 \rightarrow R_2$ indicates using R_1 to derive R_2 rules. We report both LHS and LHS-RHS derivability, separated by commas. The numbers in parentheses are times in seconds. "-" indicates that the derivability test could not be completed due to Herbie's unsound rules (Section 6.2.1). We integrate these rules for end-to-end runs of Herbie [Panchekha et al. 2015] and Megalibm [Briggs and Panchekha 2022] (Section 6.2.1).

Domain	Enumo LOC	# Enumo (Time)	# Herbie	Enumo → Herbie (Time)	Herbie → Enumo (Time)
Exponential	186	40 (4.94)	82	28.0% (0.02), 36.6% (0.02)	92.5% (0.99), 100% (0.04)
Rational	82	129 (276.16)	87	70.1% (54.77), 73.6% (55.19)	-, -
Trig	165	17 (879.65)	45	6.7% (0.0), 6.7% (0.0)	47.1% (0.01), 47.1% (0.01)

tan with the fast-forwarding algorithm described in Section 5. The following are examples of rules with cis and i: $\text{cis}(a+b) \leftrightsquigarrow \text{cis}(a) \cdot \text{cis}(b)$, $\text{cis}(0) \leftrightsquigarrow 1$, and $i \cdot i \leftrightsquigarrow -1$.

Similarly, the logarithmic, power, square root, and cube root functions are all defined in terms of e^x, the real exponential function: $e^{\log(x)} \leftrightsquigarrow x$, $a^b \leftrightsquigarrow e^{b \cdot \log(a)}$, $\sqrt{a} \leftrightsquigarrow e^{1/2 \cdot \log(a)}$, and $\sqrt[3]{a} \leftrightsquigarrow e^{1/3 \cdot \log(a)}$. As in the trigonometric case, to bootstrap fast-forwarding, we used a set of 141 automatically generated arithmetic rules and 13 handwritten rules involving the real exponential function. For both the exponential and trigonometric domains, Enumo produced the set of prior rules via methods described in previous sections.

Herbie. Herbie [Panchekha et al. 2015] is a widely used, open-source tool for improving the accuracy of floating-point expressions. Given a mathematical expression over real numbers, it synthesizes a more accurate floating-point implementation using a variety of techniques, including equality saturation. Herbie's equality saturation-based optimization pass uses a set of 358 expert-written rewrite rules to explore many programs that are equivalent over the reals, keeping only those that have lower floating-point error. Herbie's rewrite rules include many algebraic identities about rational arithmetic, trigonometry, and exponents.

Results. First, we wrote Enumo programs to synthesize boolean, rational, trigonometric (fast-forwarded), and exponential (fast-forwarded) rules for Herbie. We show a summary of these results in Table 3. Then, based on suggestions from Herbie's developers, we filtered the Herbie benchmark suite to 176 representative benchmarks taken from a variety of domains, including graphics, mathematics, and numerical analysis. In addition, we disabled polynomial approximation to isolate the effects of equality saturation within Herbie. We ran Herbie on the benchmarks under six different configurations:

- `Herbie`: Herbie's default configuration.
- `Enumo`: Enumo's rules, including boolean, rational, trigonometric, and exponential rules.
- `Enumo-Ru`: Enumo's rules with its rational rules replaced by Ruler's rational rules. This includes Enumo's boolean, trigonometric, and exponential rules, as well as Ruler's rational rules.
- `Enumo-FF`: Enumo's rules without fast-forwarded rules.
- `Enumo-R`: Enumo's rules, including only boolean and rational rules.
- `Ruler`: Ruler's ([Nandi et al. 2021]) rules for the rational and boolean domains. Ruler does not support the trigonometric and exponential domains.

We used the default node limit of 8000 nodes in Herbie's underlying equality saturation engine, i.e., upon hitting the limit, the engine stops applying the simplification rules. On 6 benchmarks, Herbie did not finish within 300 seconds; we discarded these. For all four configurations, we ran Herbie on 30 seeds.

Figure 8 shows the results of running Herbie with one boxplot for each ruleset configuration. The left plot measures the accuracy of the results using Herbie's "bits of error" metric, measuring

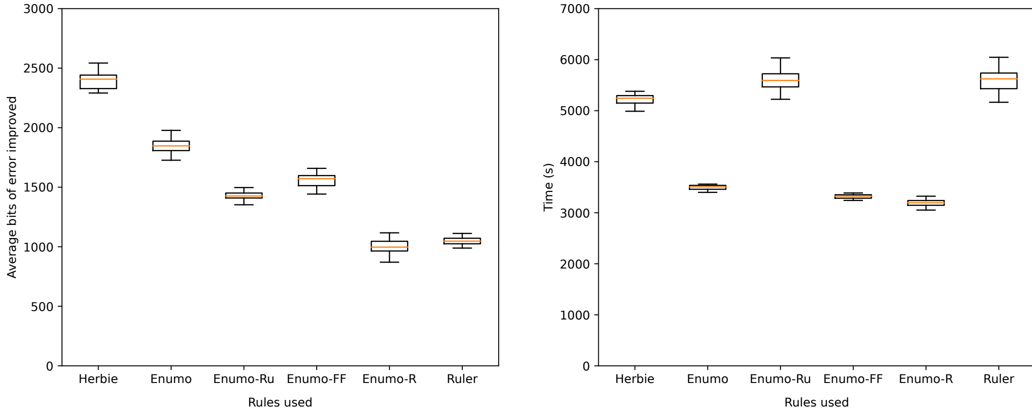


Fig. 8. Comparison of different rules on Herbie's end-to-end performance for six different configurations: (1) Herbie's default rules (`Herbie`), (2) Enumo's rules (`Enumo`), (3) Enumo's rules with its rational rules replaced by Ruler's rational rules (`Enumo-Ru`), (4) Enumo's rules without fast-fowarded rules (`Enumo-FF`), (5) Enumo's rational rules (`Enumo-R`), and (6) Ruler's rules (`Ruler`). The two plots show (*Left*) Herbie's metric for measuring accuracy (higher is better); and (*Right*) Herbie's running time (lower is better). Each boxplot represents the results from 30 seeds, where each data point is obtained by summing the values (average error, time) over all 176 benchmarks. Enumo's rules allow Herbie to improve error significantly more than Ruler's rules.

the number of "incorrect" bits in the binary representation of the floating-point result against a high-precision oracle. Using Enumo's rules, Herbie achieved 128% higher accuracy than with Ruler's rules. The right plot shows the average running time of Herbie, with rules from Enumo consistently outperforming Ruler's rules. Herbie's handwritten ruleset (Herbie) finds the most accurate programs, followed by Enumo's rules. There are two key takeaways from this experiment. It (1) demonstrates the value of fast-forwarding, and (2) shows that fast-forwarding and guided search combined lead to better rulesets than exhaustive synthesis alone.

Disabling trigonometric and exponential rules (Enumo-FF) *but leaving the rules needed to bootstrap fast-forwarding* significantly decreases accuracy, showing that fast-forwarding is necessary in the face of resource limits. By definition, the rulesets from both Enumo and Enumo-FF have equal proving power, but Figure 8 demonstrates a significant advantage for Enumo over Enumo-FF in practice. Using only the rational rules (`Enumo-R`) also lowers accuracy, showing that trigonometric and exponential rewrite rules are important for Herbie.

To our surprise, Ruler's rational rules (`Ruler`) outperformed Enumo's (`Enumo-R`). However, we show that when combined with the rest of Enumo's ruleset (`Enumo`), Enumo found more accurate programs faster than Ruler. Replacing Enumo's rational rules with Ruler's (`Enumo-Ru`) yielded a significantly worse ruleset: the combination of Enumo's rational ruleset and Enumo's trigonometric and exponential rules let Herbie to fix a wider range of floating-point errors. We suspect that Enumo's rational rules (`Enumo-R`) explore a larger space than Ruler's (causing Herbie to exceed resource limits faster) without any benefit, leading to a slight accuracy loss compared to Ruler's rational rules.

An example of where rules from Enumo excel over rules from Ruler is Herbie's "2cos" [8] benchmark, $\cos(x+\varepsilon) - \cos x$, which suffers from error when ε is relatively small. With Enumo's ruleset, Herbie used the essential rewrite `(cos (+ b a))` ⇝ `(- (* (cos b) (cos a)) (* (sin b)`

[8] https://github.com/herbie-fp/herbie/blob/main/bench/hamming/rearrangement.fpcore

`(sin a)))` to decompose $\cos(x + \varepsilon)$, and then eliminated cancellation using associative rules. The full rewrite is $\cos(x + \varepsilon) - \cos x \rightsquigarrow \cos x \cdot (\cos \varepsilon + -1) + \sin x \cdot (-\sin \varepsilon)$. Without trigonometric rules, Ruler cannot find this improvement.

However, Herbie still found more accurate programs with its handwritten ruleset: significantly, Herbie often relied on unsound division, trigonometry, and exponentiation rules to eliminate sources of errors, such as cancellation without checking if such transformation is correct for all arguments. For example, for the benchmark "2sin" [9], Herbie rewrites $\sin(x + \varepsilon) - \sin x$ to $\cos x \cdot \sin \varepsilon + (\sin \varepsilon^2 \cdot \sin x)/(-1 - \cos \varepsilon)$ using a series of rewrites that included the unsound factoring rule `(+ a b)` ⇝ `(/ (- (* a a) (* b b)) (- a b))`, which is invalid when `a` equals `b`. In contrast, Enumo generated a guarded factoring rule that included a check that `(− a b) ≠ 0` Unfortunately, Herbie is not designed to leverage conditional rules. Instead, with Enumo, we can reify the conditional guard syntactically within the rewrite itself. Herbie can directly apply such rules, e.g., `(+ a b)` ⇝ `(if (- a b) (/ (- (* a a) (* b b)) (- a b)) (+ a b))`, relying on other rules to simplify the condition syntactically. Herbie's use of unsound rules and lack of support for conditional rules presents a significant challenge in closing the gap between Herbie's handwritten and Enumo's generated rules.

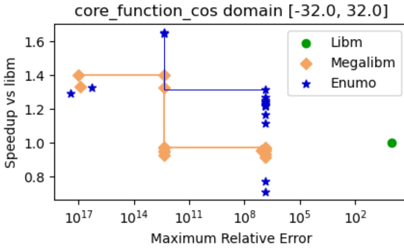
Megalibm. Next, we show how the ruleset inferred using Enumo for Herbie is also useful for Megalibm [Briggs and Panchekha 2022], another equality saturation tool that relies on numeric rewrites. Given a transcendental operator, e.g., *cos*, Megalibm synthesizes a set of low-level implementations that make different speed vs. accuracy tradeoffs. A core phase in Megalibm is using equality saturation to discover various identities over such operators.

Using sound rules from rational, trig, and exponential Enumo programs (Table 3), we ran the Megalibm benchmarks for *sin*, *cos*, and *tan*. We detail the results for *cos*, where the Megalibm baseline found the following identities: $\cos(x) = \cos(x + 4\pi)$, $\cos(x) = \cos(x + \pi + \pi)$, $\cos(x) = 2 \cdot 2 \cdot \cos(-x)\ /\ 4$, and $\cos(x) = (\pi + \pi) - ((\pi + \pi) - \cos(-x))$. The third and fourth identities are equivalent; both can be simplified to $\cos(x) = \cos(-x)$. Similarly, the second identity is just two applications of the first. Thus, the baseline yielded only 2 unique identities, from which Megalibm generated 4 implementations with differing speed and accuracy. With Enumo-generated rewrite rules, Megalibm produced the following identities: $\cos(x) = \cos(x+(\pi+\pi))$, $\cos(x) = \cos(-x)\cdot 2/2$, $\cos(x) = \cos(\pi-x)-(cos(\pi-x)\cdot 2)$, $\cos(x) = \cos(\pi-x)\cdot(-(cos(\pi-x))^0)$, and $\cos(x) = -\cos(\pi-x)$. Here, the third, fourth, and fifth identities are equivalent, yielding 3 unique identities, which Megalibm used to find 5 unique implementations. For *tan*, Megalibm used Enumo-generated rules to find *new* identities of *tan*, where the baseline ruleset did not derive any.

Figure 9 shows Megalibm's estimates of the speed vs. accuracy tradeoffs for implementations of *cos* generated by the Enumo-generated and manually developed baseline rulesets. The table summarizes how, across *sin*, *cos*, and *tan*, rules from Enumo always produced more unique identities, which typically also led to more unique implementations except for *sin*, where the baseline yielded two extra implementations. Enumo's rulesets can be applied across different tools in related domains and perform as well or better than manually developed expert rulesets.

6.2.2 Geometric Domain. Szalinski [Nandi et al. 2020] is an equality saturation-driven tool that shrinks 3D CAD (Computer-Aided Design) programs by performing rewrites over a language called *Caddy*. *Caddy* expresses CAD programs with primitives for constructive solid geometry (e.g., `Cube`, `Scale`, `Union`), basic rational arithmetic, various list constructors, and inverse transformations. Szalinski shrinks *Caddy* programs using two rulesets: a set of CAD identities and a set of custom procedural rewrite rules that discover opportunities to use inverse transformations.

[9] https://github.com/herbie-fp/herbie/blob/main/bench/hamming/rearrangement.fpcore



Fn	Megalibm		Enumo	
	Unique Impls	Unique Ids	Unique Impls	Unique Ids
sin	7	2	5	3
cos	4	2	5	3
tan	0	0	3	1

Fig. 9. Megalibm analysis. (*Left*) The pareto curve shows the implementations Megalibm found for cosine over the interval [-32.0, 32.0], with results normalized to the GNU libm implementations. Points up and to the right are better (faster and less error). Uniqueness is judged by clusters of performance. (*Right*) The number of unique identities and implementations generated with Megalibm's original rules and Enumo-synthesized rules. Notably, Megalibm found no identities or implementations for *tan*, but Enumo did.

```
1   def iter_szalinski(n):
2     lang = { (AFFINE VEC SOLID) (Cube VEC) (Cylinder VEC) (Sphere SCALAR) }
3     return iter_metric(lang, "SOLID", Depth, n)
4             .plug("VEC", { (Vec 0 0 0) (Vec 1 1 1) (Vec a a a) (Vec a b c)
5                           (Vec d e f) (Vec (BOP a d) (bop b e) (bop c f)) })
6             .plug("AFFINE", { Scale Trans })
7             .plug("SCALAR", { a 1 })
8             .plug("BOP", { + * / })
9
10  cad_idents = []
11  for i in [2, 3]:
12      wkld = iter_szalinski(i)
13      chosen = fast_forward(wkld, frep_rules, cad_idents)
14      cad_idents.extend(chosen)
```

Fig. 10. An Enumo program for learning CAD identities. `iter_szalinski` is a function that constructs workloads for *Caddy* expressions.

We focus on synthesizing the CAD identities, which helps Szalinski expose hidden structure in its input programs. Synthesizing these identities using traditional rule inference algorithms requires a full CAD interpreter, which is prohibitively difficult, and therefore unsupported by prior rule inference tools. Instead, leveraging Enumo's fast-forwarding algorithm, we present the first automatically synthesized CAD rules.

Synthesizing CAD Identities. Solid geometry can be represented mathematically via a function representation (F-rep). An F-Rep is a function $f(x, y, z)$ that interprets an arithmetic expression over x, y, and z as the geometric solid defined where f is positive [Pasko et al. 1995]. For example, the unit sphere can be represented by $1 - x^2 - y^2 - z^2$.

We use Enumo to synthesize a set of CAD identities by fast-forwarding from a small set of 5 translational rules from CAD to F-Rep combined with 15 rules over the F-Rep domain. Our prior rules for F-Rep included 6 rules over rationals and 9 substitution rules over operators that allowed

it to express *Caddy* transformations. Here are two examples of these rules:

$$Sphere(r) \rightsquigarrow 1 - \left(\frac{x}{r}\right)^2 - \left(\frac{y}{r}\right)^2 - \left(\frac{z}{r}\right)^2 \qquad Scale([w,h,l],e) \rightsquigarrow e\left[x \mapsto \frac{x}{w}\right]\left[y \mapsto \frac{y}{h}\right]\left[z \mapsto \frac{z}{l}\right]$$

Using Enumo's guided search and fast-forwarding, we synthesized a set of 20 large CAD identities of up to 13 atoms. Figure 10 shows the workload we used.

Table 4. End-to-end evaluation of Szalinski (Table 2 from Nandi et al. [2020]) using CAD identities from Enumo. The inputs are flat CAD programs from Reincarnate [Nandi et al. 2018]. For each set of identities, we show the percentages by which the initial program's AST sizes decreased (higher is better), and we show in parentheses the output AST sizes (lower is better). `No Identities`, `Szalinski's Identities`, and `Enumo-synthesized Identities` show the results of running Szalinski without CAD identities, with all identities enabled, and with Enumo-synthesized CAD identities in place of the original ones, respectively. Enumo's CAD identities exactly matched the performance of Szalinski's on 5 of 10 inputs.

Program Id	No Identities	Szalinski's Identities	Enumo-synthesized Identities
TackleBox	79% (60)	91% (26)	85% (41)
SDCardRack	72% (57)	87% (26)	86% (28)
SingleRowHolder	84% (31)	92% (16)	**92% (16)**
CircleCell	61% (31)	80% (16)	**80% (16)**
CNCBitCase	88% (27)	93% (15)	93% (16)
CassetteStorage	81% (27)	89% (15)	**89% (15)**
RaspberryPiCover	90% (27)	96% (12)	**96% (12)**
ChargingStation	81% (27)	89% (15)	82% (25)
CardFramer	52% (83)	76% (42)	**76% (42)**
HexWrenchHolder	90% (31)	95% (16)	84% (52)
Average	80% (40.1)	90% (19.9)	87% (26.3)

Results. We evaluated Szalinski's performance on a set of benchmarks from Nandi et al. [2020]'s Table 2 [10]. The results are shown in Table 4. Using Enumo's synthesized rules for CAD identities, Szalinski shrunk input benchmarks by 87% on average, while the original, handwritten rules shrunk input benchmarks by 90% on average. Upon closer inspection, we found that a subset of the Enumo-synthesized rules matched the performance of Szalinski's handwritten CAD identities. There are some rules in the Enumo ruleset that cannot be derived from Szalinski's CAD identities. However, we posit that these additional rules are not very useful for the Szalinski benchmark tests, so their presence in the Enumo ruleset leads to worse performance at low node limits.

Table 4 shows that Enumo's CAD identities closely matched the performance of the handwritten CAD identities. Fast-forwarding is effective at synthesizing rules for constructive geometry, a domain containing rewrite rules with term sizes that cannot be exhaustively enumerated, for which writing an interpreter is challenging, and which prior work did not support.

6.3 Ruleset Manipulation with Enumo

This section presents a case study that leverages Enumo's operators for ruleset manipulation. Here, we are interested in synthesizing rewrite rules over 4-bit bitvectors (BV4) and transforming those rules into a usable ruleset over larger bitvectors. Synthesizing rules for small bitvectors is very fast because there are relatively few possible values in the domain. Rules that work for small bitvectors are likely, but not guaranteed, to be valid for large bitvectors, as well. In this case study, we start

[10] Nandi et al. [2020] mentions that these benchmarks were decompiled using the Reincarnate [Nandi et al. 2018] mesh decompiler.

Table 5. Comparison of rule synthesis for different widths of bitvectors. Shown for each bitvector width are (i) the number of rules generated from an Enumo program (time in seconds) for that domain, (ii) the number of Enumo-synthesized BV4 rules that are valid in that domain (time in seconds), and (iii) the percentage of the generated rules that are derivable from the validated BV4 rules (both LHS and LHS-RHS derivability).

Domain	Generated Rules (Time)	Valid BV4 Rules (Time)	Validated → Generated
BV8	230 (32.78)	230 (3.19)	(100%, 100%)
BV16	236 (85.50)	224 (8.36)	(97%, 97%)
BV32	232 (191.74)	224 (10.81)	(98%, 99%)
BV64	250 (431.70)	220 (16.97)	(93%, 93%)
BV128	190 (1784.14)	210 (38.68)	(90%, 91%)

by synthesizing BV4 rules using the same Enumo program described in Section 6.1. Then, we use Enumo to "cast" the rules into the domain of larger bitvectors (BV8, BV16, BV32, BV64, and BV128). Finally, we validate the rules in the new domain to find the subset of sound BV4 rules that are still sound for larger bitvectors. We compare these rules to rules that were synthesized directly in the larger bitvector domains using the same Enumo program as we used to synthesize BV4 rules. The results of this case study are shown in Table 5. Validating BV4 rules was much faster than synthesizing rules from scratch (38 seconds vs. 29 minutes for BV128) and still produced a useful ruleset: across all bitvector sizes, the validated BV4 rules retained at least 90% of the proving power of the directly generated rules; for 8-bit bitvectors, the validated BV4 rules had equal proving power. This case study highlights the usefulness of Enumo's abstractions— though porting rulesets from one domain to another was not a design consideration in the development of Enumo, its operators support this use-case without modification.

7 DISCUSSION, LIMITATIONS, AND FUTURE WORK

Scheduling. Equality saturation engines mitigate the effects of rule ordering by non-destructively applying all rules in each iteration of the algorithm. However, in practical applications where saturation is unlikely, only a finite number of iterations of the equality saturation algorithm complete before terminating due to resource limits. Deciding *which* rules to run *when* becomes critical, and splitting rules into batches can dramatically alter the results. Based on preliminary experiments, we find that certain additional strategies significantly improve the results of equality saturation tools. Two of these include using operators like compress and using a *saturating scheduler*, which iteratively (1) applies saturating rules (is_saturating in Figure 4) to saturation, then (2) applies the other rules for a single iteration. Developing scheduling strategies requires a more systematic investigation of scheduling techniques than this paper provides, but we are excited to further explore rule scheduling in the context of equality saturation.

Conditional Rewrites. We have also only partially explored conditional rule inference in Enumo. To use Enumo for inferring state-of-the-art rules in more complex domains like LLVM IR, robust conditional rule inference as well as program analyses to satisfy side conditions will be necessary. We leave this as future work.

Overfitting. It is possible to write an Enumo workload that **overfits** in order to find certain rules. For example, to find the rule `(+ (+ ?c ?a) (- ?b ?c))` $\rightsquigarrow$ `(+ ?b ?a)`, one could construct an Enumo workload with *only* the terms `((+ (+ z x) (- y z)))` and `(+ y c)`, i.e., for a rule $\ell \rightsquigarrow r$, a workload representing only ℓ and r instantiated with concrete variables. However, this requires the user to know a priori exactly which rules they want, which is rarely the case. Good Enumo programs must strike a balance between sufficiently narrowing the search space so as to make rule inference feasible without overly constraining the workload. In practice, most of our Enumo

programs are significantly shorter (i.e., fewer lines of code) than simply writing the rules by hand; we have not found overfitting to be a problem in the domains we have explored. On the other hand, the ability to write an overfit workload is potentially useful: if the overfit workload *does not* find the target rule, it might indicate that the target rule is unsound. This can allow Enumo users to interactively to explore a domain.

Beyond Rule Synthesis. While this paper used Enumo to synthesize rewrite rules for and by equality saturation, the DSL itself is generic and not restricted to such applications. Enumo lays the groundwork for future applications in bounded model checking and sketch-guided synthesis. For example, Enumo could be useful in axiom synthesis tools such as LAS [Krogmeier et al. 2022a], where the enumeration order greatly affects the quality of results.

8 RELATED WORK

Prior work has used e-graphs for rule inference [Nandi et al. 2021; Nötzli et al. 2019]. [Nötzli et al. 2019] uses enumerative synthesis to infer axioms for the CVC4 theorem prover. Ruler [Nandi et al. 2021] outperformed Nötzli et al. [2019] in various domains. In this paper, we show that Enumo can outperform Ruler in terms of both scalability and generality of domains (Section 6).

Similarly, theory exploration is a well-studied topic, focusing on eagerly synthesizing lemmas that may be useful for verification tasks. A recent tool in this space is TheSy [Singher and Itzhaky 2021], which performs inductive theory exploration using equality saturation and symbolic values to efficiently filter candidate conjectures. TheSy's key insight is to leverage congruence closure to implement an induction prover within the equality saturation framework. Similar tools for theory exploration use random testing to find potential candidates [Claessen et al. 2013, 2010]. IsaCoSy [Johansson et al. 2010] synthesizes inductive theorems for the Isabelle theorem prover [Paulson 1986]. To keep the search space of terms tractable, IsaCoSy selectively enumerates only terms that are not reducible from existing rules. A similar technique is used by Ta et al. [2017] in lemma synthesis for proving entailments with separation logic. We believe the abstractions provided by Enumo for guided search and ruleset manipulation can be used to scale lemma synthesis in these tools. In future work, we would like to express the inductive prover from TheSy in Enumo.

Many custom tools synthesize rewrite rules in specific domains. [Xu et al. 2023] proposed a tool that synthesizes rewrite rules for quantum circuit optimization. Jia et al. [2019] developed a tool synthesizing graph substitutions for deep neural networks. RuleSy [Butler 2019] uses a combination of synthesis and specification mining to find proof rules representing the mined specification. Wang et al. [2022] presented an equality saturation-driven tool for learning query rewrite rules. In contrast, Enumo's DSL-based approach is not specialized to any particular domain; our evaluation in Section 6 shows that Enumo works across diverse domains. Prior work used machine learning to assist in rewrite rule inference [Krogmeier et al. 2022b; Singh and Solar-Lezama 2016]; in particular, Singh and Solar-Lezama [2016] supports some forms of conditional rules. This paper shows that by using Enumo's novel term enumeration primitives, rule inference scales to support grammars that have conditional operators; however, full support for conditional rule inference is left for future work.

Finally, several tools have focused on automatically inferring peephole optimizations [Bansal and Aiken 2006; Buchwald 2015; Davidson and Fraser 2004; Menendez and Nagarakatte 2017] and instruction selection [Buchwald et al. 2018]. Two major challenges with these optimizations are the presence of side conditions and their large grammars. This paper shows that with Enumo's guided enumeration strategy, it is possible to find rewrite rules with side conditions. We also show that it is possible to scale to large grammars, like that of Halide [Ragan-Kelley et al. 2013]. We will

continue to explore techniques for more general conditional rewrite rule inference, and we are excited to use Enumo to infer more optimizations for frameworks like LLVM.

The Enumo DSL is designed to facilitate efficient term enumeration given a grammar. Effective enumeration has been explored in many other contexts, like relational algebra, sorting algorithms, testing, and generating well-typed lambda terms [Abiteboul et al. 1995; Christiansen et al. 2016; Duregård et al. 2012; Flajolet and Salvy 1995; Rodriguez Yakushev and Jeuring 2010]. In the most closely related work, Duregård et al. [2012] propose *Feat*, a Haskell library for composing enumerations. They use a lazy mechanism (*functional enumeration*) to scale enumeration and leverage memoisation to efficiently index into a stream of enumerated terms. In a previous prototype of Enumo, we explored a similar mechanism, but we found that in the context of rewrite rule inference, e-graph size is the bottleneck, not enumeration time. Therefore, in our final prototype, we use a simpler method for materializing a workload into a concrete set of terms. A similarity between the Enumo DSL and the Feat library is the idea of composable workloads. Duregård et al. [2012] define a set of combinators that let them compose smaller enumerations effectively. As Section 4 showed, Enumo workloads can be composed using a set of operators (Plug, Filter, Union), some of which are similar to Feat (e.g., *union*ing two workloads or enumerations). A key feature of Enumo is that it evaluates a workload only when converting it to an e-graph (as shown in Section 4); this lets Enumo leverage a unique set of operators like plug and filter to optimize a workload before it is evaluated and converted to an e-graph (see examples in Section 4).

9 CONCLUSION

This paper presents Enumo, a new domain-specific language for rewrite rule inference using equality saturation. Enumo offers novel term enumeration primitives and exposes useful ruleset operators that enable incremental, composable, workload-guided rewrite rule inference. We also introduce a new *fast-forwarding* algorithm for generating rewrite rules; fast-forwarding finds rewrite rules for domains not supported by prior tools. Enumo subsumes the capabilities of state-of-the-art tools for rule inference [Nandi et al. 2021; Nötzli et al. 2019] in terms of ruleset quality and scalability. Several case-studies demonstrate that small, modular Enumo programs generate useful rulesets that can be plugged in to existing equality saturation tools or composed to quickly find rulesets across diverse domains. Enumo lets users strategically guide the rule inference process at a high level and incrementally build effective rulesets.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their thoughtful feedback. We are grateful to Ian Briggs and Pavel Panchekha for helping us run Herbie and Megalibm using Enumo's rules.

This material is based upon work supported by an Amazon Research Award,the National Science Foundation under Grant Nos. 1749571 and 2232339 as well as the DARPA V-SPELLS program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, Department of Defense, or the U.S. Government.

REFERENCES

Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley. http://webdam.inria.fr/Alice/

Sorav Bansal and Alex Aiken. 2006. Automatic Generation of Peephole Superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 394–403. https://doi.org/10.1145/1168857.1168906

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (Snowbird, UT) *(CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 171–177.

Hans-J. Boehm. 2020. Towards an API for the Real Numbers. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 562–576. https://doi.org/10.1145/3385412.3386037

Ian Briggs and Pavel Panchekha. 2022. Synthesizing Mathematical Identities with E-Graphs. In *Proceedings of the 1st ACM SIGPLAN International Symposium on E-Graph Research, Applications, Practices, and Human-Factors* (San Diego, CA, USA) *(EGRAPHS 2022)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3520308.3534506

Sebastian Buchwald. 2015. Optgen: A Generator for Local Optimizations. In *Compiler Construction*, Björn Franke (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 171–189.

Sebastian Buchwald, Andreas Fried, and Sebastian Hack. 2018. Synthesizing an Instruction Selection Rule Library from Semantic Specifications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. Association for Computing Machinery, New York, NY, USA, 300–313. https://doi.org/10.1145/3168821

Eric Butler. 2019. *Automatic Generation of Procedural Knowledge Using Program Synthesis.* Ph. D. Dissertation. University of Washington, USA. https://hdl.handle.net/1773/43656

David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. 2023. Babble: Learning Better Abstractions with E-Graphs and Anti-Unification. *Proc. ACM Program. Lang.* 7, POPL, Article 14 (jan 2023), 29 pages. https://doi.org/10.1145/3571207

Jan Christiansen, Nikita Danilenko, and Sandra Dylus. 2016. All Sorts of Permutations (Functional Pearl). *SIGPLAN Not.* 51, 9 (sep 2016), 168–179. https://doi.org/10.1145/3022670.2951949

Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. 2013. Automating Inductive Proofs Using Theory Exploration. In *Automated Deduction – CADE-24*, Maria Paola Bonacina (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 392–406.

Koen Claessen, Nicholas Smallbone, and John Hughes. 2010. QuickSpec: Guessing Formal Specifications Using Testing. In *Tests and Proofs*, Gordon Fraser and Angelo Gargantini (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 6–21.

Coq. 2022. Library Coq.PArith.BinPos. https://coq.inria.fr/library/Coq.PArith.BinPos.html.

Samuel Coward, George A. Constantinides, and Theo Drane. 2022. Abstract Interpretation on E-Graphs. arXiv:2203.09191 [cs.LO]

Samuel Coward, George A. Constantinides, and Theo Drane. 2023. Automating Constraint-Aware Datapath Optimization using E-Graphs. arXiv:2303.01839 [cs.AR]

Jack W. Davidson and Christopher W. Fraser. 2004. Automatic Generation of Peephole Optimizations. *SIGPLAN Not.* 39, 4 (apr 2004), 104–111. https://doi.org/10.1145/989393.989407

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2007. Efficient E-Matching for SMT Solvers. In *CADE*.

David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: A Theorem Prover for Program Checking. *J. ACM* 52, 3 (May 2005), 365–473. https://doi.org/10.1145/1066100.1066102

Jonas Duregård, Patrik Jansson, and Meng Wang. 2012. Feat: Functional Enumeration of Algebraic Types. In *Proceedings of the 2012 Haskell Symposium* (Copenhagen, Denmark) *(Haskell '12)*. Association for Computing Machinery, New York, NY, USA, 61–72. https://doi.org/10.1145/2364506.2364515

Philippe Flajolet and Bruno Salvy. 1995. Computer Algebra Libraries for Combinatorial Structures. *J. Symb. Comput.* 20, 5–6 (nov 1995), 653–671. https://doi.org/10.1006/jsco.1995.1070

Cheng Fu, Hanxian Huang, Bram Wasti, Chris Cummins, Riyadh Baghdadi, Kim Hazelwood, Yuandong Tian, Jishen Zhao, and Hugh Leather. 2023. Q-Gym: An Equality Saturation Framework for DNN Inference Exploiting Weight Repetition. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Chicago, Illinois) *(PACT '22)*. Association for Computing Machinery, New York, NY, USA, 291–303. https://doi.org/10.1145/3559009.3569673

Zachary Grannan, Niki Vazou, Eva Darulova, and Alexander J. Summers. 2022. REST: Integrating Term Rewriting with Program Verification. In *36th European Conference on Object-Oriented Programming (ECOOP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 13:1–13:29. https://doi.org/10.4230/LIPIcs.ECOOP.2022.13

Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York,

NY, USA, 47–62. https://doi.org/10.1145/3341301.3359630

Moa Johansson, L. Dixon, and A. Bundy. 2010. Conjecture Synthesis for Inductive Theories. *Journal of Automated Reasoning* 47 (2010), 251–289.

Moa Johansson, Dan Rosén, Nicholas Smallbone, and Koen Claessen. 2014. Hipster: Integrating Theory Exploration in a Proof Assistant. In *Intelligent Computer Mathematics,* Stephen M. Watt, James H. Davenport, Alan P. Sexton, Petr Sojka, and Josef Urban (Eds.). Springer International Publishing, Cham, 108–122.

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

Thomas Koehler, Philip W. Trinder, and Michel Steuwer. 2021. Sketch-Guided Equality Saturation: Scaling Equality Saturation to Complex Optimizations in Languages with Bindings. *ArXiv* abs/2111.13040 (2021).

Dexter Kozen. 1977. Complexity of Finitely Presented Algebras. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing* (Boulder, Colorado, USA) *(STOC '77)*. Association for Computing Machinery, New York, NY, USA, 164–177. https://doi.org/10.1145/800105.803406

Paul Krogmeier, Zhengyao Lin, Adithya Murali, and P. Madhusudan. 2022a. Synthesizing Axiomatizations Using Logic Learning. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 185 (oct 2022), 29 pages. https://doi.org/10.1145/3563348

Paul Krogmeier, Zhengyao Lin, Adithya Murali, and P. Madhusudan. 2022b. Synthesizing Axiomatizations Using Logic Learning. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 185 (oct 2022), 29 pages. https://doi.org/10.1145/3563348

Jedidiah McClurg, Miles Claver, Jackson Garner, Jake Vossen, Jordan Schmerge, and Mehmet E. Belviranli. 2021. Optimizing Regular Expressions via Rewrite-Guided Synthesis. https://doi.org/10.48550/ARXIV.2104.12039

David Menendez and Santosh Nagarakatte. 2017. Alive-Infer: Data-Driven Precondition Inference for Peephole Optimizations in LLVM. *SIGPLAN Not.* 52, 6 (jun 2017), 49–63. https://doi.org/10.1145/3140587.3062372

Chandrakana Nandi, James R. Wilcox, Pavel Panchekha, Taylor Blau, Dan Grossman, and Zachary Tatlock. 2018. Functional Programming for Compiling and Decompiling Computer-Aided Design. *Proc. ACM Program. Lang.* 2, ICFP, Article 99 (jul 2018), 31 pages. https://doi.org/10.1145/3236794

Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing Structured CAD Models with Equality Saturation and Inverse Transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 31–44. 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, Article 119 (oct 2021), 28 pages. https://doi.org/10.1145/3485496

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

Julie L. Newcomb, Steven Johnson, Shoaib Kamil, Andrew Adams, and Ratislav Bodik. 2020. Verifying and Improving Halide's Term Rewriting System with Program Synthesis. *Proceedings of the ACM on Programming Languages* OOPSLA (2020).

Andres Nötzli, Andrew Reynolds, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Clark Barrett, and Cesare Tinelli. 2019. Syntax-Guided Rewrite Rule Enumeration for SMT Solvers. In *Theory and Applications of Satisfiability Testing – SAT 2019*, Mikoláš Janota and Inês Lynce (Eds.). Springer International Publishing, Cham, 279–297.

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

Alexander Pasko, Valery Adzhiev, Alexei Sourin, and Vladimir Savchenko. 1995. Function representation in geometric modeling: concepts, implementation and applications. *The visual computer* 11 (1995), 429–446.

L C Paulson. 1986. Natural Deduction as Higher-Order Resolution. *J. Log. Program.* 3, 3 (oct 1986), 237–258. https://doi.org/10.1016/0743-1066(86)90015-4

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* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1066–1082. https://doi.org/10.1145/3385412.3386001

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

Alexey Rodriguez Yakushev and Johan Jeuring. 2010. Enumerating Well-Typed Terms Generically. In *Approaches and Applications of Inductive Programming*, Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 93–116.

Rohit Singh and Armando Solar-Lezama. 2016. Swapper: A Framework for Automatic Generation of Formula Simplifiers Based on Conditional Rewrite Rules. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design* (Mountain View, California) *(FMCAD '16)*. FMCAD Inc, Austin, Texas, 185–192.

Zak Singh. 2022. Deep Reinforcement Learning for Equality Saturation. *University of Cambridge* (2022). https://www.cl.cam.ac.uk/~ey204/pubs/MPHIL_P3/2022_Zak.pdf.

Eytan Singher and Shachar Itzhaky. 2021. Theory Exploration Powered by Deductive Synthesis. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 125–148.

Quang-Trung Ta, Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin. 2017. Automated Lemma Synthesis in Symbolic-Heap Separation Logic. *Proc. ACM Program. Lang.* 2, POPL, Article 9 (dec 2017), 29 pages. https://doi.org/10.1145/3158097

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. ACM, New York, NY, USA, 264–276. https://doi.org/10.1145/1480881.1480915

Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 2021. Vectorization for Digital Signal Processors via Equality Saturation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 874–886. https://doi.org/10.1145/3445814.3446707

Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. 2020. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *Proceedings of the VLDB Endowment* (2020).

Zhaoguo Wang, Zhou Zhou, Yicun Yang, Haoran Ding, Gansen Hu, Ding Ding, Chuzhe Tang, Haibo Chen, and Jinyang Li. 2022. WeTune: Automatic Discovery and Verification of Query Rewrite Rules. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 94–107. https://doi.org/10.1145/3514221.3526125

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* POPL.

Amanda Xu, Abtin Molavi, Lauren Pick, Swamit Tannu, and Aws Albarghouthi. 2023. Synthesizing Quantum-Circuit Optimizers. In *Proceedings of the 44st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Orlando, Florida) *(PLDI 2023)*. Association for Computing Machinery. https://doi.org/10.1145/3591254

Yichen Yang, Phitchaya Mangpo Phothilimtha, Yisu Remy Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. In *Proceedings of Machine Learning and Systems*. arXiv:2101.01332

Received 2023-04-14; accepted 2023-08-27