Rewrite Rule Inference Using Equality Saturation

CHANDRAKANA NANDI*, University of Washington, USA
MAX WILLSEY*, University of Washington, USA
AMY ZHU, University of Washington, USA
YISU REMY WANG, University of Washington, USA
BRETT SAIKI, University of Washington, USA
ADAM ANDERSON, University of Washington, USA
ADRIANA SCHULZ, University of Washington, USA
DAN GROSSMAN, University of Washington, USA
ZACHARY TATLOCK, University of Washington, USA

Many compilers, synthesizers, and theorem provers rely on rewrite rules to simplify expressions or prove equivalences. Developing rewrite rules can be difficult: rules may be subtly incorrect, profitable rules are easy to miss, and rulesets must be rechecked or extended whenever semantics are tweaked. Large rulesets can also be challenging to apply: redundant rules slow down rule-based search and frustrate debugging.

This paper explores how equality saturation, a promising technique that uses e-graphs to *apply* rewrite rules, can also be used to *infer* rewrite rules. E-graphs can compactly represent the exponentially large sets of enumerated terms and potential rewrite rules. We show that equality saturation efficiently shrinks both sets, leading to faster synthesis of smaller, more general rulesets.

We prototyped these strategies in a tool dubbed Ruler. Compared to a similar tool built on CVC4, Ruler synthesizes 5.8× smaller rulesets 25× faster without compromising on proving power. In an end-to-end case study, we show Ruler-synthesized rules which perform as well as those crafted by domain experts, and addressed a longstanding issue in a popular open source tool.

Additional Key Words and Phrases: Equality Saturation, Rewrite Rules, Program Synthesis

1 INTRODUCTION

Rewrite systems transform expressions by repeatedly applying a given set of rewrite rules. Each rule $\ell \rightsquigarrow r$ rewrites occurrences of the syntactic pattern ℓ to instances of another *semantically equivalent* pattern r. Rewrite systems are effective because they combine individually simple rules into sophisticated transformations, maintain equivalence between rewritten expressions, and are easy to extend by adding new rules.

Many compilers, program synthesizers, and theorem provers rely on rewrite systems [Detlefs et al. 2005; Hoe and Arvind 2000; Peyton Jones et al. 2001]. For example, rewriting is essential for improving program analyses and code generation [Blindell 2013; Chen et al. 2018; Lattner et al. 2021; Ragan-Kelley et al. 2013] and for automating verification [Barrett et al. 2011; Bertot and Castran 2010; De Moura and Bjørner 2008; Nipkow et al. 2002]. Without rule-based simplification, Halide-generated code can suffer 26× slowdown [Newcomb et al. 2020] and the Herbie floating-point synthesizer [Panchekha et al. 2015] can return 10× larger programs.

Several noteworthy projects have developed tool-specific techniques for checking or inferring rules [Bansal and Aiken 2006; Joshi et al. 2002; Menendez and Nagarakatte 2017; Singh and Solar-Lezama 2016], but implementing a rewrite system still generally requires domain experts to first manually develop rulesets by trial and error. Such slow, ad hoc, and error-prone approaches hinder design space exploration for new domains and discourage updating existing systems.

*Both authors contributed equally to this work.

PL, 2021, USA
.

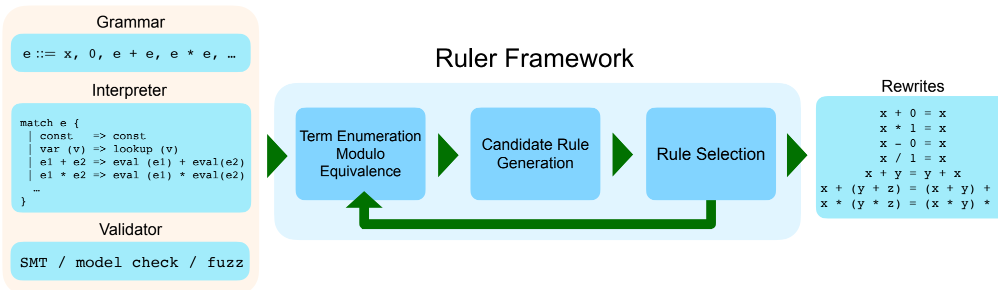


Fig. 1. *Ruler Workflow.* Given a grammar and interpreter for a target domain, Ruler uses e-graphs and equality saturation to efficiently enumerate potential rewrite rules and iteratively select a small set of general, orthogonal rules. Ruler supports various validation strategies to ensure soundness and speed up synthesis, including constraint solving (e.g., SMT), model checking, and fuzzing (Section 6).

To address these challenges, we propose a simple, domain-general approach that uses equality saturation [Tate et al. 2009; Willsey et al. 2021] as a rewrite system *on the domain of rewrite rules themselves* to quickly synthesize effective rulesets.

In the past, tool-specific techniques to iteratively infer rewrite rules have implicitly adopted a common three-step approach, each constructing or maintaining a set:

(1) Enumerate terms from the given domain to build the *term set* T.
(2) Select candidate rules from $T \times T$ to build the *candidate set* C.
(3) Filter C to select a sound set of useful rules to build the *rule set* R.

We identify and abstract this workflow to provide generic rule inference for user-specified domains.

Our key insight is that *what makes equality saturation successful in rewrite rule application is also useful for rule inference*. Equality saturation can simultaneously prove many pairs of terms equivalent with respect to a given ruleset. Our approach uses equality saturation to shrink the set T of enumerated terms (lowering candidate *generation* cost) by merging terms equivalent under R, and to shrink the set C of candidate rules (lowering candidate *selection* cost) by removing rules derivable by R. Thus, it uses the set R of rewrite rules to rewrite the next batch of candidate rewrite rules *even as R is being synthesized.*

We prototyped these insights in a tool dubbed Ruler (Figure 1). Compared to a state-of-the-art rule synthesizer [Nötzli et al. 2019] built into the CVC4 theorem prover [Barrett et al. 2011], Ruler synthesizes smaller rulesets in less time without reducing the set of derivable equivalences. We demonstrate how Ruler can generate expert-quality rulesets by using it to replace all of Herbie's rules for rational numbers, uncovering missing rules that resolved a known bug in Herbie.

In summary, this paper's contributions include:

- A novel rule synthesis algorithm that uses e-graphs [Nelson 1980] to compactly encode large sets of terms and equality saturation to efficiently filter and minimize rulesets (Section 3).
- A generic implementation of this algorithm within the Ruler[1] rewrite rule inference framework that synthesizes rules for user-specified domains given a grammar and its interpreter.
- A comparison against a recent CVC4-based rule synthesizer that shows Ruler synthesizes 5.8× smaller rulesets 25× faster without compromising the deriving power of the rulesets.

[1]Ruler will be made open-source and publicly available at `[link redacted for review]`.

- A case study demonstrating that, in an end-to-end application of a real world tool, Ruler's automatically generated rulesets are as good as manually-crafted expert rules (Section 5).

The rest of the paper is organized as follows. Section 2 provides background on rewrite systems, e-graphs, and equality saturation. Section 3 presents Ruler's core algorithm, Section 4 evaluates our implementation of Ruler against a CVC4-based rule synthesizer. Section 5 presents a case study where Ruler-generated rules can competitively replace expert-crafted rules in the Herbie numerical program synthesizer. Section 6 provides several empirical analyses of Ruler's algorithm and compares different instantiations, verification back-ends, and rule discovery strategies. Section 7 discusses limitations of our approach and opportunities for future work. Section 8 presents related work, and Section 9 concludes.

2 BACKGROUND

To build a rewrite system for a target domain, programmers must develop a set of rewrite rules and then use a rewrite engine to apply them, e.g., for optimization, synthesis, or verification. Ruler helps automate this process using e-graphs to compactly represent sets of terms and using equality saturation to filter and minimize candidate rules.

2.1 Implementing Rewrite Systems

Developing Rewrite Rules. Within a given domain D, a rewrite rule $\ell \leftrightsquigarrow r$ is a first-order formula consisting of a single equation, where ℓ and r are terms in D and all free variables are $\forall$-quantified. Rewrite rules must be sound: for any substitution σ of their free variables, ℓ and r must have the same semantics, i.e., $[\![\sigma(\ell)]\!]_D = [\![\sigma(r)]\!]_D$.

In many cases, rewrite rules must also be engineered to meet (meta)constraints of the rewrite engine responsible for applying them. For example, classic term rewriting approaches often require special considerations for cyclic (e.g., $(x+y) \rightsquigarrow (y+x)$) or expansive (e.g., $x \rightsquigarrow (x+0)$) rules [Baader and Nipkow 1998]. The choice of rules and their ordering can also affect the quality and performance of the resulting rewrite system [Whitfield and Soffa 1997]. Different ruleset variations may cause a rewrite system to be faster or slower and may be able to derive different sets of equivalences.

To a first approximation, smaller rulesets of more general, less redundant rules are desirable. Having fewer rules speeds up rule-based search since there are fewer patterns to repeatedly match against. Having more general, orthogonal rules also increases a rewrite system's "proving power" by expanding the set of equivalences derivable after a smaller number of rule applications. Avoiding redundancy also aids debugging, making it possible to diagnose a misbehaving rule-based search or optimization by eliminating one rule at a time.

Automatic synthesis aims to generate rulesets that are sound and that include non-obvious, profitable rules that even domain experts may overlook for years.[2] Ideally, ruleset synthesis itself should also be fast; rapid rule inference can help programmers explore the design space for rewrite systems in new domains. It can also help with rewrite system maintenance since rulesets must be rechecked and potentially extended whenever any operator for a domain is added, removed, or updated, i.e., when the semantics for the domain evolves.

Applying Rules with Rewrite Engines. Given a set of rewrite rules, a rewrite engine is tasked with either optimizing a given term into a "better" equivalent term (e.g., for peepholes [McKeeman 1965]

[2] https://github.com/halide/Halide/pull/3719
https://github.com/uwplse/herbie/issues/261
https://github.com/apache/tvm/pull/5974
https://github.com/Z3Prover/z3/issues/2575
https://github.com/Z3Prover/z3/pull/4663

or superoptimization [Massalin 1987]) or proving two given terms equivalent, i.e., solving the *word problem* [Bezem et al. 2003].

Classic term rewriting systems destructively update terms as they are rewritten. This approach is generally fast, but complicates support for cyclic or expansive rules, and makes both rewriting performance and output quality dependent on fine-grained rule orderings. Past work has extensively investigated how to mitigate these challenges by scheduling rules [Barendregt et al. 1987; Borovanskỳ et al. 1998; Dershowitz 1982; Knuth and Bendix 1983], special casing cyclic and expansive rules [Bachmair et al. 2000; Dershowitz 1987; Eker 2003; Lucas 2001], and efficiently implementing rewrite rule-based search [Clavel et al. 2007; Kirchner 2015; Visser 2001a,b]. Many systems still rely on ad hoc rule orderings and heuristic mitigations developed through trial and error, though recent work [Newcomb et al. 2020] has demonstrated how reduction orders [Baader and Nipkow 1998] can be automatically synthesized and then used to effectively guide destructive term rewriting systems.

2.2 E-graphs

An *equality graph* (e-graph) is a data structure commonly used in theorem provers [Detlefs et al. 2005; Joshi et al. 2002; Nelson 1980] to efficiently compute a congruence relation over a set of terms. E-graph implementations are backed by union-find data structures [Tarjan 1975]. An e-graph consists of a set of *e-classes* (equivalence classes) where each e-class contains one or more *e-nodes*, and each e-node is a tuple of a function symbol and a list of children e-classes.

An e-graph compactly stores an equivalence (more specifically, congruence) relation over terms. We say that an e-graph *represents* the terms in its equivalence relation, recursively defined as:

- An e-graph represents all terms represented by any of its e-classes.
- An e-class represents all terms represented by any of its e-nodes. Terms represented by the same e-class are considered equivalent.
- An e-node $f(c_1, c_2, ...)$ represents a term $f(t_1, t_2, ...)$ iff the function symbols match and each e-class c_i represents term t_i. Terms represented by the same e-node are equivalent and have the same top-level function symbol.

E-graphs use hashconsing [Ershov 1958] to ensure e-nodes are never duplicated in an e-graph. This sharing helps keep e-graphs compact even when representing exponentially many terms. E-graphs may even contain cycles, in which case they represent an infinite set of terms.

Figure 2 shows two small e-graphs. In the left example, each e-node is in its own e-class. This e-graph represents only the term $a + a$. In the right example, the top e-class contains two *equivalent* e-nodes: the e-node with the function symbol + represents the term $a + a$, and the e-node with the function symbol $*$ represents the term $a * 2$.

2.3 Equality Saturation

Equality saturation [Tate et al. 2009] is a promising technique that repurposes e-graphs to implement efficient rewrite engines. Starting from a term t, equality saturation builds an initial e-graph E representing t, and then repetedly applies rules to expand E into a large set of equivalent terms.

For optimization or synthesis, a user-provided cost function determines the "best" term t_b also represented by t's e-class; t_b is then extracted and returned as the optimized result. For proving an equivalence $t_1 = t_2$, both t_1 and t_2 are initially added to E and the equality saturation engine tests whether their e-classes ever merge.

A major advantage of equality saturation engines is that they are non-destructive: information is never lost when applying a rule. This enables support for cyclic and expansive rules and mitigates many scheduling challenges: rule ordering generally does not affect the quality of the final result.

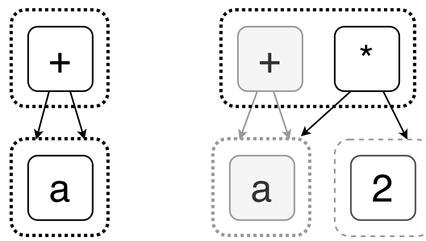


Fig. 2. Initially, the e-graph (left) represent only the term $a + a$. The rewrite rule $(x + x) \leftrightsquigarrow (x \times 2)$ is applied to the initial e-graph which merges the term $a \times 2$ into the *same* e-class that contains $a + a$. The dotted boxes show e-classes and the solid boxes show e-nodes.

```
def eq_sat(t, R):
    egraph = make_egraph(t)
    saturated = false
    while (not saturated) && (not timeout()):
        saturated = true
        for ℓ ⇝ r in R:
            for (σ, c_ℓ) in egraph.search(ℓ):
                c_r = egraph.add(σ(r))
                if not egraph.same_eclass(c_ℓ, c_r)
                    egraph.union(c_ℓ, c_r)
                    saturated = false
    return egraph.extract(cost_fun)
```

Fig. 3. Equality Saturation. Initially, the e-graph only represents the AST of the original input term. Semantics-preserving rewrite rules are applied until the e-graph saturates or a timeout is reached. A cost function guides the extraction of the “best” program from the e-graph.

Recent work has applied high performance equality saturation libraries [Willsey et al. 2021] to produce state-of-the-art synthesizers and optimizers across several diverse domains [Nandi et al. 2020; Panchekha et al. 2015; Premtoon et al. 2020; VanHattum et al. 2021; Wang et al. 2020; Wu et al. 2019; Yang et al. 2021].

Figure 3 shows the core equality saturation algorithm. Given a term t and a set of rewrite rules R, the eq_sat procedure first makes an initial e-graph corresponding to t's abstract syntax tree (AST); each e-class initially contains a single e-node. Rewrite rules are then repeatedly applied to all terms represented in the e-graph. To apply a rule $\ell \rightsquigarrow r$, the algorithm first uses a procedure called *e-matching* [Detlefs et al. 2005] to search all represented terms for those matching ℓ. This returns a list of pairs (σ, c_ℓ) of substitutions and the corresponding e-class where the rule matched some term $\sigma(\ell)$. Each substitution σ is then *applied* to r, yielding a new term $\sigma(r)$ equivalent to $\sigma(\ell)$. If $\sigma(r)$ is not already represented by the same e-class $\sigma(l)$ is, then $\sigma(r)$ is added to a fresh e-class c_r, and c_r is unioned (merged) with c_ℓ, the e-class where $\sigma(\ell)$ was matched. This process illustrates how rewriting in an e-graph is non-destructive: applying rules only adds information.

```
1  def ruler (iterations):
2    T = empty_egraph()
3    R = {}
4    for i ∈ [0, iterations]:
5      # add new terms directly to the e-graph representing T
6      add_terms(T, i)
7      loop:
8        # combine e-classes in the e-graph representing T that R proves equivalent
9        run_rewrites(T, R)
10       C = cvec_match(T)
11       if C = {}:
12         break
13       # choose_eqs only returns valid candidates by using 'is_valid' internally
14       # and it filters out all invalid candidates from C
15       R = R ∪ choose_eqs(R, C)
16   return R
```

Fig. 4. Ruler's Core Algorithm. The iterations parameter determines the maximum number of connectives in the terms Ruler will enumerate.

The second e-graph in Figure 2 shows an example of e-graph rewrite rule application. Upon applying the rewrite rule $(a + a) \rightsquigarrow (a \times 2)$ to the e-graph on the left, a new term is added to the e-graph. This new term is in the same e-class as the original term, indicating they are equivalent. One edge from this new e-node points to the e-class containing x, while the other points to a freshly-added e-class containing the e-node for 2.

When applying rules no longer adds new information (i.e., all terms and equalities were already present), we say the e-graph has *saturated*. Upon saturation (or timeout), a user-provided cost function determines which represented term to *extract* from the e-graph. Equality saturation supports various extraction procedures, from simply minimizing AST size to more sophisticated procedures based on integer linear programming or genetic algorithms [Willsey et al. 2021].

3 RULER

This section describes Ruler, a new equality saturation-based rewrite rule synthesis technique. Like other rule synthesis approaches, Ruler iteratively performs three steps:

(1) Enumerate terms into a set T.
(2) Search $T \times T$ for a set of candidate equalities C.
(3) Choose a useful, valid subset of C to add to the ruleset R.

Ruler's core insight is that e-graphs and equality saturation can help compactly represent the sets T, C, and R, leading to a faster synthesis procedure that produces smaller rulesets R with greater proving power (Section 4.1.2).

3.1 Ruler Overview

Figure 4 shows Ruler's core synthesis algorithm, which is parameterized by the following:

- The number of iterations to perform the search for (line 4);

- The language grammar, given in the form of a term enumerator (add_terms, line 6), which takes the number of variables or constants to enumerate over;
- The procedure for validating candidate rules, is_valid (called inside choose_eqs, Figure 5 line 20).

These parameters provide flexibility for supporting different domains, making Ruler a rule synthesis framework rather than a single one-size-fits-all tool. Ruler uses an e-graph to compactly represent the set of terms T. In each iteration, Ruler first extends the set T with additional terms from the target language. Each term $t \in T$ is tagged with a *characteristic vector* (cvec) that stores the result of evaluating t given many different assignments of values to variables.

After enumerating terms, Ruler uses equality saturation (run_rewrites) to merge terms in T that can be proved equivalent by the rewrite rules already discovered (in the set R); Next, Ruler computes a set C of candidate rules (cvec_match). It finds pairs $(t_1, t_2) \in T \times T$ where t_1 and t_2 are from distinct e-classes but have matching cvecs and thus are likely to be equivalent. Thanks to run_rewrites, no candidate in C should be derivable from R. However, C is often still large and contains many redundant or invalid candidate rules.

Finally, Ruler's choose_eqs procedure picks a valid subset of C to add to R, ideally finding the smallest extension which can establish all equivalences implied by $R \cup C$. Ruler tests candidate rules for validity using a domain-specific is_valid function. This process is repeated until there are no more equivalences to learn between terms in T, at which point Ruler begins another iteration. We detail each of these phases in the rest of this section.

3.2 Enumeration Modulo Equivalence

Rewrite rules encode equivalences between terms, often as relatively small "find and replace" patterns. Thus, a straightforward strategy for finding candidate rules is to find all equivalent pairs of terms up to some maximum size. Unfortunately, the set of terms up to a given size grows exponentially, making complete enumeration impractical for many languages. This challenge may be mitigated by biasing enumeration towards "interesting" terms, e.g. drawn from important workloads, or by avoiding bias and using sampling techniques to explore larger, more diverse terms. Ruler can support both domain-specific prioritization and random sampling via the add_terms function. While these heuristics can be very effective, they often risk missing profitable candidates for new classes of inputs or use cases.

Term space explosion can also be mitigated by partitioning terms into equivalence classes and only considering a single, canonical representative from each class. Similar to partial order reduction techniques in model checking [Peled 1998] this can make otherwise intractable enumeration feasible. Ruler defaults to this complete enumeration strategy, using an e-graph to compactly represent T and equality saturation to keep T partitioned with respect to equivalences derivable from the rules in R even as they are being discovered.

Enumerating Terms in an E-graph. E-graphs are designed to represent large sets of terms efficiently by exploiting sharing and equivalence. For sharing, e-graphs maintain deduplication and maximal reuse of subexpressions via hash-consing. If some term a is already represented in an e-graph, checking membership is constant time and adding it again has no effect. The first time $(a + a)$ is added, a new e-class is introduced with only a single e-node, representing the + with both operands pointing to a's e-class. If $((a + a) * (a + a))$ is then added, a new e-class is introduced with only a single e-node, representing the $*$ with both operands pointing to $(a + a)$'s e-class. Thus as Ruler adds expressions to T, only the new parts of each added expression increase the size of T in memory.

On iteration i, calling add_terms(T, i) adds all (exponentially many) terms with i connectives to the e-graph. The first iteration calls add_terms with an empty e-graph to add terms with $i = 0$

connectives, thus specifying how many variables and which constants (if any) will be included in the search space. Since these terms are added to an e-graph, deduplication and sharing automatically provide efficient representation, but they do not, by themselves, provide an equivalence reduction to help avoid enumerating over many equivalent terms.

Compacting T using R. Ruler's e-graph not only stores the set of terms T, but also an equivalence relation (more specifically, a congruence relation) over those terms. Since the children of an e-node are e-classes, a single e-node can represent exponentially many equivalent terms. Initially, the e-graph stores no equivalences, i.e., each term is in its own equivalence class.

As the algorithm proceeds, Ruler learns rules and places them in the set R of accepted rules.[3] At the beginning of its inner loop (line 9), Ruler performs equality saturation with the rules from R. Equality saturation will unify classes of terms in the e-graph that can be proven equivalent with rules from R. To ensure that run_rewrites only *shrinks* the term e-graph, Ruler performs this equality saturation on a copy of the e-graph, and then copies the newly learned equalities (e-class merges) back to the original e-graph. This avoids polluting the e-graph with terms added during equality saturation, i.e., it prevents enumeration of new terms based on intermediate terms introduced during equality saturation.

Ruler's inner loop only terminates once there are no more rules to learn, so the next iteration (add_terms, line 6) *only enumerates over the canonical representatives* from the equivalence classes of terms with respect to R that have been represented up to that point. This compaction of the term space makes complete enumeration possible for non-trivial depths and makes Ruler much more efficient in finding a small set of powerful rules. Section 6 demonstrates how compaction of T is essential to Ruler's performance.

Since R may contain rules that use partial operators, Ruler's equality saturation implementation only merges e-classes whose cvecs agree in at least one non-null way (see the definition of *match* in Section 3.3). For example, consider that $x/x \leftrightsquigarrow 1 \in R$, and both $\frac{a+a}{a+a}$ and $\frac{a-a}{a-a} \in T$. The pattern x/x matches both terms, but equality saturation will not merge $\frac{a-a}{a-a}$ with 1, since $\frac{a-a}{a-a}$ is never defined. On the other hand, $\frac{a+a}{a+a}$ can merge with 1 since the cvecs match.

Prior work [Nötzli et al. 2019] on rule inference applies multiple filtering passes to minimize rule sets *after* they are generated. These filters include subsumption order, variable ordering, filtering modulo alpha-renaming, and removing rules in the congruence closure of previously found rules. Ruler eliminates the need for such filtering using equality saturation on the e-graph representing T. Since enumeration takes place over e-classes in T, equivalent terms are "pre-filtered" automatically.

3.3 Candidate Rules

Given a set (or in Ruler's case, an e-graph) of terms T, rewrite rule synthesis searches $T \times T$ for pairs of equivalent terms that could potentially be a rule to add to R. The set of candidate rules is denoted C.

The naive procedure for producing candidate rules simply considers every distinct pair:

$$C = \{l \leftrightsquigarrow r \mid l, r \in T.\ l \neq r \wedge \forall \sigma.\ l[\sigma] = r[\sigma]\}$$

This is prohibitively expensive for two main reasons. First, it will produce many rules that are either in or can be proven by the existing ruleset R. In fact, the naive approach should always produce supersets of C from previous iterations; accepting a candidate rule from C into R would not prevent it from being generated in R in the following iteration. Second, most of the candidates will be unsound, and sending too many unsound candidates to choose_eqs burdens it unnecessarily, since it must search C for valid candidates by invoking the user-supplied is_valid procedure. Ruler's use of

[3]While Figure 4 shows R starting empty, the user may instead initialize R with trusted axioms if they choose.

an e-graph to represent the term set T addresses both of the these inefficiencies with techniques called *canonical representation* and *characteristic vectors*.

Canonical Representation. Consider a situation where $(x + y) \leftrightsquigarrow (y + x) \in R$ and both $(a + b)$ and $(b + a)$ are in T. When selecting terms from which to build a candidate rule, considering both $(a+b)$ and $(b+a)$ would be redundant; any rules derived from one could be derived from the other by composing it with commutativity of +. In some rewriting systems, this composition of rewrites cannot be achieved since cyclic rules like commutativity are not permitted. Equality saturation, however, handles and in many cases prefers such compositional rules, since it results in fewer rules to search over the e-graph.

To prevent generating candidate rules which are already derivable by the rules in R, Ruler only considers a single term from each e-class when building candidate rules. When searching for candidate rules, Ruler considers only term pairs (l, r) where $l \neq r$ and both are canonical representatives of e-classes in T. This ensures candidate rules cannot be derived from R; if they could have been, then l and r would have been in the same e-class after the call to run_rewrites.

Characteristic Vectors. Canonical representation reduces C from $T \times T$ to $T' \times T'$ where T' is the set of canonical terms from T, but it does not prevent a full $O(n^2)$ search of $T' \times T'$ for valid candidate rules. Ruler employs a technique called *characteristic vectors* (cvecs) to prevent this quadratic search by only considering pairs that are *likely* valid. Ruler associates a characteristic vector v_i with each e-class i. The cvec is the result of evaluating t_i, the canonical term in e-class i, over a set of inputs that serves as a "fingerprint"[4] for the value of that e-class. Stated precisely, let σ_j for $j \in [1, m]$ be a predetermined family of m mappings from variables in T to concrete values, and let eval be the evaluator for the given language. The cvec for e-class i is:

$$v_i = [\mathsf{eval}(\sigma_j, t_i) \mid j \in [1, m]]$$

Ruler computes cvecs incrementally and without redundancy during enumeration using an *e-class analysis* [Willsey et al. 2021] to associate a cvec with each e-class; let i be an e-class, t_i its canonical term, and v_i its cvec:

- when $t_i = n$ for a constant n, v_i is populated by copies of n;
- when $t_i = f(t_{j_1}, \ldots, t_{j_n})$ for some n-ary operator f from the given language, v_i is computed by mapping f over the cvecs of the subterms: $v_i = \mathsf{map}(f, \mathsf{zip}(v_{j_1}, \ldots, v_{j_n}))$
- when $t_i = x$ for a variable x, v_i is populated by values from the target domain; choosing values to populate the cvecs of variables can be done randomly or with a domain-specific approach (Section 6 compares two approaches).

To support partial operators (e.g., division), cvecs may have a null value in them to indicate failure to evaluate. We say that cvecs *match* if their non-null values agree in every (and at least one) position, i.e., cvecs $[a_1, \ldots, a_n]$ and $[b_1, \ldots, b_n]$ match iff:

$$\forall i.\ a_i = b_i \lor a_i = \mathsf{null} \lor b_i = \mathsf{null} \quad \text{and} \quad \exists i.\ a_i = b_i \land a_i \neq \mathsf{null} \land b_i \neq \mathsf{null}$$

When e-classes in the e-graph representing T merge, they will have matching cvecs, because they have been proven equivalent by valid rules. Ruler aborts if cvecs of merging e-classes do not match; empirically, this helps avoid learning unsound rules even when is_valid is not sound (Section 6.2).

Section 4 and Section 5 discuss how cvecs are generated for different domains. Characteristic vectors serve as a filter for validity: if i, j are e-classes and v_i does not match v_j, (using the definition

[4] Section 8 discusses prior work [Bansal and Aiken 2006; Jia et al. 2019] that uses "fingerprints" for synthesizing peephole optimizations and graph substitutions.

of match from Section 3.1) then $t_i \leftrightsquigarrow t_j$ is not valid. This allows Ruler to not consider those pairs when building C:

$$C = \{t_i \leftrightsquigarrow t_j \mid i, j \in \text{e-classes of } T.\ \mathsf{match}(v_i, v_j)\}$$

The cvec_match procedure (called at Figure 4, line 10) constructs C by grouping e-classes from T based on their cvecs and then taking pairs of canonical terms from each of those groups.

Validation. The candidate set C contains rules that are likely, but not guaranteed, to be valid. The choose_eqs function (discussed in Section 3.4) must validate these before returning them by using the user-supplied is_valid function. The soundness of Ruler's output, i.e., whether every rule in R is valid, depends on the soundness of the provided is_valid procedure. Many rule synthesis implementations [Jia et al. 2019; Singh and Solar-Lezama 2016] use SMT solvers to perform this validation. Ruler supports arbitrary validation procedures: small domains may use model checking, larger domains may use SMT, and undecidable domains may decide to give up a guarantee of soundness and use a sampling-based validation. Section 6.2 compares validation techniques for different domains.

3.4 Choosing Rules

After finding a set of candidate rules C, Ruler selects a valid subset of rules from C to add to the rule set R using the choose_eqs procedure (Figure 4, line 15). As long as choose_eqs returns a valid, non-empty subset of C, Ruler's inner loop will terminate: the number of e-classes with matching cvecs (i.e., the subset of T used to compute C) decreases in each iteration since R is repeatedly extended with rules that will cause new merges in run_rewrites. Ideally, choose_eqs quickly finds a minimal extension of R that enables deriving all equivalences implied by $R \cup C'$ where C' is the valid subset of C. choose_eqs also removes invalid candidates from C; if it returns the empty set (i.e., none of the candidates in C are valid), then Ruler's inner loop will terminate in the next iteration due to line 11 in Figure 4.

The candidate rules in C are not derivable by R, but many of the candidate rules may be able to derive each other, especially in the context of R. For example, the following candidate set is composed of three rules from the boolean domain,[5] and any two can derive the third:

```
(^ x x) = false          (& x false) = false          (& x false) = (^ x x)
```

An implementation of choose_eqs that only returns a single rule $c \in C$ avoids this issue, since adding c to R prevents those rules derivable by $R \cup \{c\}$ from being candidates in the next iteration of the inner loop. However, a single-rule implementation will be slow to learn rules, since it can only learn one at a time (Table 1 of our evaluation shows there are sometimes thousands of rules to learn). Additionally, such an implementation has to decide which rule to select, ideally picking the "strongest" rules first. For example, if $a, b \in C$ and $R \cup \{a\}$ can derive b but $R \cup \{b\}$ can not derive a, then selecting b before a would be a mistake, causing the algorithm to incur an additional loop.

Ruler's implementation of choose_eqs, shown in Figure 5, is parameterized by a value n with default of ∞. At $n = 1$, choose_eqs simply returns a single valid candidate from C. For higher n, choose_eqs attempts to return a list of up to n valid rules all at once. This can speed up Ruler by requiring fewer trips around its inner loop, but risks returning many rules that can derive each other. To mitigate this, choose_eqs tries to not choose rules that can derive each other. In its main loop (line 14), choose_eqs uses the select function to pick the *step* best rules from C according to a syntactic heuristic.[6] Ruler then validates the selected rules and adds them to a set K of "keeper" rules which it will ultimately return. It then employs the shrink procedure (line 34) to eliminate

[5] ^ here represents XOR

[6] Ruler's syntactic heuristic prefers candidates with the following characteristics (lexicographically): more distinct variables, fewer constants, shorter larger side (between the two terms forming the candidate), shorter smaller side, and fewer distinct operators.

```
1  # R is the accepted ruleset so far, C is the candidate ruleset.
2  # Ruler's implementation of choose_eqs is based on a more flexible choose_eqs_n.
3  def choose_eqs(R, C, n = ∞):
4      for step ∈ [100, 10, 1]:
5          if step ≤ n:
6              C = choose_eqs_n(R, C, n, step)
7      return C
8
9  # n is the number of rules to choose from C, and step is a granularity parameter.
10 # A larger step size allows you to eliminate redundant rules faster.
11 def choose_eqs_n(R, C, n, step):
12     # let K be the list of "keepers" which we will return
13     K = []
14     while C ≠ ∅:
15         # pick the best step candidate rules from C according to a heuristic
16         # that approximates rule "generality", including subsumption.
17         C_best, C = select(step, C)
18
19         # add the valid ones to K
20         K = K ∪ {c | c ∈ C_best. is_valid(c)}
21
22         # remember all the invalid candidates in a global variable bad;
23         # Ruler uses this to prevent known-invalid candidates from entering C again (not shown)
24         bad = bad ∪ {c | c ∈ C_best. ¬is_valid(c)}
25
26         # stop if we have enough rules
27         if |K| ≥ n:
28             return K[0..n]
29
30         # try to prove terms remaining in C equivalent using rules from R ∪ K
31         C = shrink(R ∪ K, C)
32     return K
33
34 def shrink(R, C):
35     E = empty_egraph()
36     for (l ⇝ r) ∈ C:
37         E = add_term(E, l)
38         E = add_term(E, r)
39     E = run_rewrites(E, R)
40     # return the extracted versions of rules from C, leaving out anything that was proven equivalent
41     return {extract(E,l) ⇝ extract(E,r) | (l ⇝ r) ∈ C. ¬equiv(E,l,r)}
```

Fig. 5. Ruler's implementation of choose_eqs, which aims to minimize the candidate set C by eliminating subsets that the remainder can derive.

candidates from C that can be derived be $R \cup K$. This works similarly to run_rewrites in the Ruler algorithm, but shrink works over the remaining *candidate set* C instead of T.

Ruler's choose_eqs invokes the inner choose_eqs_n procedure with increasingly small step sizes (*step* is defined on line 4). Larger step sizes allow shrink to quickly "trim down" C when it contains many candidates. However, a large step also means that choose_eqs may admit *step* rules into K at once, some of which may be able to prove each other. Decreasing the step size to 1 eliminates this issue.

Ruler uses $n = \infty$ by default for maximum performance, and Section 6 measures the effects of this choice on Ruler's performance and output.

3.5 Implementation

We implemented Ruler in Rust using the egg [Willsey et al. 2021] e-graph library for equality saturation. By default Ruler uses Z3 [De Moura and Bjørner 2008] for SMT-based validation, although using other validation backends is simple (Section 6).

Ruler's core consists of under 1,000 lines of code, allowing it to be simple, extensible, and generic over domains. Compared to the rewrite synthesis tool inside the CVC4 solver [Barrett et al. 2011; Nötzli et al. 2019], Ruler is an order of magnitude smaller. Since Ruler's core algorithm does not rely on SMT, Ruler can learn rewrite rules over domains unsupported by SMT-LIB [Barrett et al. 2016], or for alternative semantics for those domains [7] (Section 6.3).

In the following sections, we provide various evaluations of three representative domains on top of Ruler's core. Each domain highlights a verification back-end and cvec generation strategy Ruler supports:

- booleans and bitvector-4: these are small domains which Ruler can efficiently model check and generate sound rules by construction — the cvecs are complete.
- bitvector-32: demonstrates that Ruler supports SMT-based verification for large, non-uniform domains.
- rationals: demonstrates that random sampling is adequate for larger but continuous domains. This domain also showcases Ruler's support for partial operators like division.

The implementation of booleans, bitvectors, and rationals are in approximately 100, 400, and 300 lines, respectively.

4 EVALUATION

In evaluating Ruler, we are interested in the following research questions:

- *Performance.* Does Ruler synthesize rewrite rules quickly compared to similar approaches?
- *Compactness.* Does Ruler synthesize small rulesets?
- *Derivability.* Do Ruler's rulesets derive the rules produced by similar approaches?
- *End-to-End.* How well do Ruler's synthesized rules perform compared to rules generated by experts in real applications?
- *Sensitivity Analysis.* How do the different components of Ruler's core algorithm affect performance and the size of the synthesized ruleset?
- *Validation Analysis.* How do different validation strategies affect Ruler's output?

We first evaluate performance, compactness, and derivability (Section 4.1) by comparing Ruler against recent work [Nötzli et al. 2019] that added rewrite rule synthesis to the CVC4 SMT solver [Barrett et al. 2011]. We compose Ruler's rules with Herbie [Panchekha et al. 2015] to show an end-to-end evaluation (Section 5). We then study the effects of different choices in Ruler's search algorithm and the different validation and cvec generation strategies (Section 6).

[7] For example, the Halide [Ragan-Kelley et al. 2013] tool uses division semantics where $x/0 = 0$; this is different from the SMT-LIB semantics, but it can easily be encoded using the `ite` operator.

Parameters		Ruler			CVC4			Ruler / CVC4	
Domain	# Conn	Time (s)	# Rules	Drv	Time (s)	# Rules	Drv	Time	Rules
bool	2	0.01	20	1	0.13	53	1	0.06	0.38
bool	3	0.06	28	1	0.82	293	1	0.07	0.10
bv4	2	0.14	49	1	4.47	135	0.98	0.03	0.36
bv4	3	4.30	272	1	372.26	1978	1	0.01	0.14
bv32	2	13.00	46	0.97	18.53	126	0.93	0.70	0.37
bv32	3	630.09	188	0.98	1199.53	1782	0.91	0.53	0.11
								0.04	0.17
								Harmonic Mean	

Table 1. Ruler tends to synthesize smaller, more powerful rulesets in less time than CVC4. The table shows synthesis results across domains and maximum term size (in number of connectives, "# Conn"). The domains are booleans, bitvector-4, and bitvector-32 each of which is evaluated for # Conn = 2 and # Conn = 3. For verification, Ruler uses model checking for booleans and bitvector-4 and Z3 for bitvector-32. The "Drv" column shows the fraction that tool's synthesized ruleset can derive of the other's ruleset; for example, the final row indicates that Ruler's 188 rules derived 98% of CVC4's 1,782 rules, and CVC's rules derived 91% of Ruler's. The final two columns show the ratios of synthesis times and ruleset sizes between the two tools.

$$
\begin{aligned}
\langle \mathit{var} \rangle &::= \; \mathrm{x} \mid \mathrm{y} \mid \mathrm{z} \\[6pt]
\langle \mathit{expr} \rangle &::= \; \langle \mathit{literal} \rangle \mid \langle \mathit{var} \rangle \\
&\mid \; \sim \langle \mathit{expr} \rangle \\
&\mid \; \langle \mathit{expr} \rangle \;\&\; \langle \mathit{expr} \rangle \\
&\mid \; \langle \mathit{expr} \rangle \;\hat{}\; \langle \mathit{expr} \rangle \\
&\mid \; (\langle \mathit{expr} \rangle \mid \langle \mathit{expr} \rangle)
\end{aligned}
$$

(a) Boolean (bool) grammar.

$$
\begin{aligned}
\langle \mathit{var} \rangle &::= \; \mathrm{x} \mid \mathrm{y} \mid \mathrm{z} \\[6pt]
\langle \mathit{expr} \rangle &::= \; \langle \mathit{literal} \rangle \mid \langle \mathit{var} \rangle \\
&\mid \; \sim \langle \mathit{expr} \rangle \mid - \langle \mathit{expr} \rangle \mid \\
&\mid \; \langle \mathit{expr} \rangle + \langle \mathit{expr} \rangle \mid \langle \mathit{expr} \rangle - \langle \mathit{expr} \rangle \mid \langle \mathit{expr} \rangle \times \langle \mathit{expr} \rangle \\
&\mid \; \langle \mathit{expr} \rangle \ll \langle \mathit{expr} \rangle \mid \langle \mathit{expr} \rangle \gg \langle \mathit{expr} \rangle \\
&\mid \; \langle \mathit{expr} \rangle \;\&\; \langle \mathit{expr} \rangle \mid (\langle \mathit{expr} \rangle \mid \langle \mathit{expr} \rangle)
\end{aligned}
$$

(b) Bitvector grammar used for both bv4 and bv32. $\sim$ is bitwise negation; unary $-$ is two's-complement. We use shift semantics where $a \ll b = a \gg b = 0$ when $b \geq \mathit{width}$.

Fig. 6. The grammars used by both Ruler and CVC4 in our evaluation.

4.1 Comparison with CVC4

Both Ruler and the CVC4 synthesizer are written in systems programming languages (Rust and C++, respectively), and both take a similar approach to synthesizing rewrite rules: enumerate terms, find valid candidates, select rules, and repeat.

At the developers' suggestion, we used CVC4 version 1.8 with –sygus-rr-synth to synthesize rules. We enabled their rule filtering techniques (–sygus-rr-synth-filter-cong, –sygus-rr-synth-filter-match, –sygus-rr-synth-filter-order). We enabled their rule checker (–sygus-rr-synth-check) to verify all synthesized rules. Additionally, we also disabled use of any pre-existing rules from CVC4 to guide the rule synthesis (using –no-sygus-sym-break, –no-sygus-sym-break-dynamic).

Table 1 shows the results of the comparison. The following text discusses the results in detail, but, in short, Ruler synthesizes smaller rulesets in less time that have more proving power (Section 4.1.2).

4.1.1 Benchmark Suite. We compare Ruler against CVC4 for booleans, bitvector-4, and bitvector-32. Figure 6 shows the grammars. Both Ruler and CVC4 are parameterized by the domain (bool, bv4, or bv32), the number of distinct variables in the grammar, and the size of the synthesized term.[8] All benchmarks were single-threaded and run on an AMD 3900X 3.6GHz processor with 32GB of RAM. Both Ruler and CVC4 were given 3 variables and no constants to start the enumeration.

4.1.2 Derivability. A bigger ruleset is not necessarily a better ruleset. We designed Ruler to minimize ruleset size while not compromising on its capability to prove equalities. We define a metric called the *deriving ratio* to compare two rulesets. Ruleset A has deriving ratio p with respect to ruleset B if set A can derive a fraction p of the rules in B ($A \vDash b$ means rule set A can prove rule b):

$$p = |B_A|/|B| \quad \text{where} \quad B_A = \{b \mid b \in B.\ A \vDash b\}$$

If A and B have deriving ratio of 1 with respect to each other, then they can each derive all of the other's rules.

We use egg's equality saturation procedure to test derivability. To test whether $A \vDash b$ (where $b = b_l \leftrightsquigarrow b_r$) we add b_l and b_r to an empty e-graph, run equality saturation using A, and check to see if the e-classes of b_l and b_r merged. We run egg with 5 iterations of equality saturation. Since this style of proof is bidirectional (egg is trying to rewrite both sides at the same time), derivations of $b_l = b_r$ can be as long as 10 rules from A.

4.1.3 Bitvector and Boolean Implementation. Ruler supports different implementations of the is_valid procedure (Section 3.3) for different domains. When the domain is small enough, Ruler can use efficient model checking. For example, there are only $(2^4)^3 = 4096$ assignments of bitvector-4 to three variables. By using cvecs of that length to capture all possibilities, Ruler can guarantee that the cvec_match procedure returns only valid candidate rules, and is_valid need not perform any additional checking. Ruler uses model checking for booleans and bitvector-4, and it uses SMT-backed verification for bitvector-32.

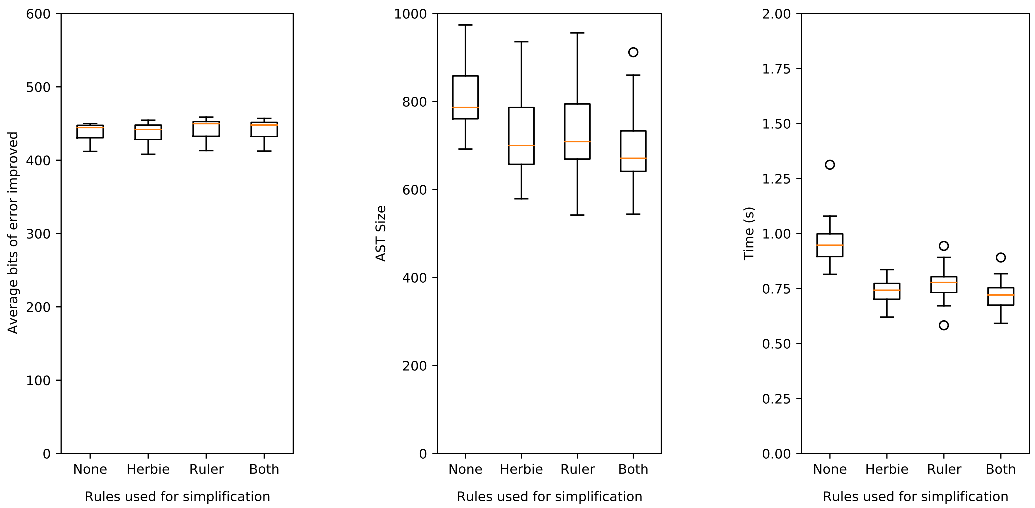
4.1.4 Results. Table 1 shows the results of our comparison with CVC4's rewrite rule synthesis. On average (harmonic mean), Ruler produces 5.8× smaller rulesets 25× faster than CVC4. Ruler and CVC4's results can derive most of each other. On the harder benchmarks (in terms of synthesis times), Ruler's results have a higher derivability ratio; they can prove more of CVC4 rules than vice-versa.

5 END-TO-END EVALUATION OF RULER: A RATIONAL CASE STUDY

How good are Ruler's rewrite rules? Can they be used with existing rewrite-based tools with little additional effort? This section demonstrates how Ruler's output can be plugged directly into an existing rewrite-driven synthesis tool, Herbie.

Herbie [Panchekha et al. 2015] is a widely-used, open-source tool for automatically improving the accuracy of floating point expressions, with thousands of users and regular yearly releases. Herbie takes as input a numerical expression and returns a more accurate expression. It is implemented in Racket [Racket 2021]. Herbie has separate phases for error localization (by sampling), series expansion, regime inference, and simplification, which work together to increase accuracy of numerical programs. The simplification phase uses algebraic rewrites to simplify mathematical expressions, thereby also enabling further accuracy improvements. These are applied using an equality saturation engine. In the past, the set of algebraic rewrites has been the cause of many bugs; of 8 open bugs at the time of this writing, six have been tagged "rules" by the developers [Herbie 2021b]. Ruler was able to find rules that addressed one of these issues.

[8] Size is measured in number of connectives, e.g., a has 0, $(a + b)$ has 1, and $(a + (b + c))$ has 2. In CVC4, this is set with the –sygus-abort-size flag.



(a) Improvement in average error, Herbie's metric for measuring accuracy (higher is better).

(b) Size of the output AST produced by Herbie (lower is better).

(c) Herbie's running time (lower is better).

Fig. 7. Comparing Herbie results between four configurations. Each boxplot represents the results from 30 seeds, where each data point is obtained by summing the value (average error, AST size, time) over all 51 benchmarks. The columns dictate what rational rules Herbie has access to: either none, its default rules, only Ruler's rules, or both. Herbie's rational rules reduce AST size and speed up simplification without reducing accuracy, and Ruler's rules perform similarly (with or without Herbie's rules).

5.1 Experimental Setup

We implemented rationals in Ruler using rational and bigint libraries in Rust [Rust 2021a,b]. We then synthesized rewrite rules over rational arithmetic and ran Herbie with the resulting ruleset.

The Herbie benchmark suite has 155 benchmarks; 55 of those are over rationals — i.e., all expressions in these benchmarks consist only of operators: $+, -, \times, /, abs, neg$. At the developers' suggestion, we filtered out 4 of the 55 benchmarks because they repeatedly timed out. We ran all our experiments on the remaining 51 benchmarks under four different configurations:

- None: remove all the rational rewrite rules from Herbie's simplification phase. Rational rules are those which consist only of rational operators and no others. Note that all other components of Herbie are left intact, including rules over rational operators combined with other operators, and rules entirely over other operators. None is the baseline.
- Herbie: no changes to Herbie, simply run it on the 51 benchmarks.
- Ruler: replace Herbie's rational rules with output of Ruler.
- Both: run Herbie with both Ruler's rational rules and the original Herbie rational rules.

Herbie has a default timeout of 180 seconds for each benchmark. It has a node limit of 5000 in its underlying equality saturation engine, i.e., it stops applying the simplification rules once the e-graph has 5000 e-nodes. We ran our experiments with three settings — (1) using the defaults, (2) increasing the timeout to 1000 seconds, and the node-limit to 10,000 to account for the addition of extra rules to Herbie's ruleset, and (3) decreasing the node limit to 2500 — we found that our results were stable and robust across all three settings. Figure 7 shows the results for the default setting.

For all four configuration (None, Herbie, Ruler, Both), we ran Herbie for 30 seeds (because Herbie's error localization relies on random sampling).

We used Ruler to synthesize rational rules of depth 2 with 3 variables using random testing for validation ("rational" under Table 2).[9] Ruler learned 50 rules in 18 seconds, all of which were proven sound with an SMT post-pass. Four rules were expansive — i.e., rules like $(a \leftrightsquigarrow (a \times 1))$ whose LHS is only a variable. We removed these expansive rules from the ruleset as per the recommendation of the Herbie developers. Herbie's rules are uni-directional — we therefore expanded our rules for compatibility, ultimately leading to 76 uni-directional Ruler rules.

5.2 Discussion

The Herbie simplifier uses equality saturation to find smaller, equivalent programs. The simplifier itself does not directly improve accuracy; rather, it generates more candidates that are then used in the other accuracy improving components of Herbie. While ideally, Herbie would return a more accurate *and* smaller output, Herbie's ultimate goal is to find more accurate expressions, even if it sacrifices AST size. Herbie's original ruleset has been developed over the past 6 years by numerical methods experts to effectively accomplish this goal. Any change to these rules must therefore ensure that it does not make Herbie's result less accurate.

Figure 7 shows the results of running Herbie with rules synthesized by Ruler. Each box-plot corresponds to one of the four configurations. The baseline (None) and Herbie in Figure 7's accuracy and AST size plots highlight the significance of rational rewrites in Herbie — these expert-written rules reduce AST size without reducing accuracy. The plots for Ruler show that running Herbie with only Ruler's rational rules has almost the same effect on accuracy and AST size as Herbie's original, expert written ruleset. The plot for Both shows that running Herbie together with Ruler's rules further reduces AST size, still without affecting accuracy. The timing plots show that adding Ruler's rules to Herbie does not make it slower. The baseline timing is slower than the rest because removing all rational simplification rules causes Herbie's other components take much longer to find the same results.

In summary, Ruler's rational rewrite rules can be easily integrated into Herbie, and they perform as well as expert-written rules without incurring any additional overhead.

Derivability. Herbie's original rational ruleset consisted of 52 rational rules. Ruler synthesized 76 uni-directional rational rules (50 bidirectional rules). We compared the two rulesets for proving power, by deriving each with the other using the approach described in Section 4.1.2. We found that Herbie's ruleset was able to derive 42 out of the 50 Ruler rules. It failed to derive the remaining 8. Ruler on the other hand, was able to derive all 52 rules from Herbie. We highlight two of the 8 Ruler rules that Herbie's ruleset failed to derive that concern multiplications interaction with absolute value: $(|a \times b| \leftrightsquigarrow |a| \times |b|)$, and $(|a \times a| \leftrightsquigarrow a \times a)$.

Fixing a Herbie Bug. The above two rules found by Ruler helped the Herbie team address a GitHub issue [Herbie 2021a]. In many cases, Herbie may generate large, complex outputs without improving accuracy, which makes the program unreadable and hard to debug. This is often due to lack of appropriate rules for expression simplification. The issue raised by a user ([Herbie 2021a]) was in fact due to the missing rule $(|x| \times |x| \leftrightsquigarrow x \times x)$. The two rules above, can together, accomplish the effect of this rule, thereby solving the issue. We submitted these two rules to the Herbie developers and they added them to their ruleset.

[9]For rationals, the add_terms implementation enumerates terms by depth rather than number connectives, since that matches the structure of Herbie's existing rules.

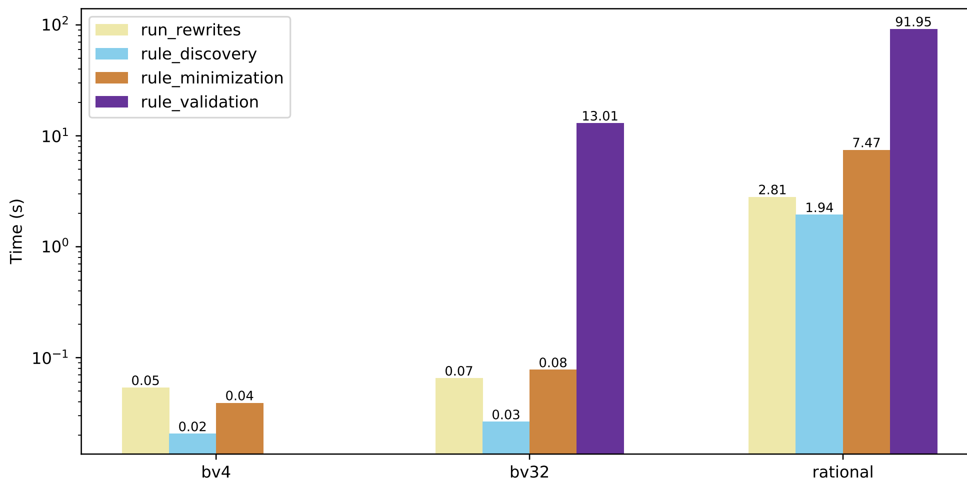


Fig. 8. Search Profile. Time spent (in log scale) by Ruler in the various phases of its algorithm for bitvector-4, bitvector-32, and rationals. Most of the time is spent in rule validation (when applicable), then minimization. Notice that bitvector-4 does not have an explicit validation time because the rules are correct by construction — the cvecs are complete.

6 SENSITIVITY STUDIES: ANALYZING RULER FRAMEWORK PARAMETERS

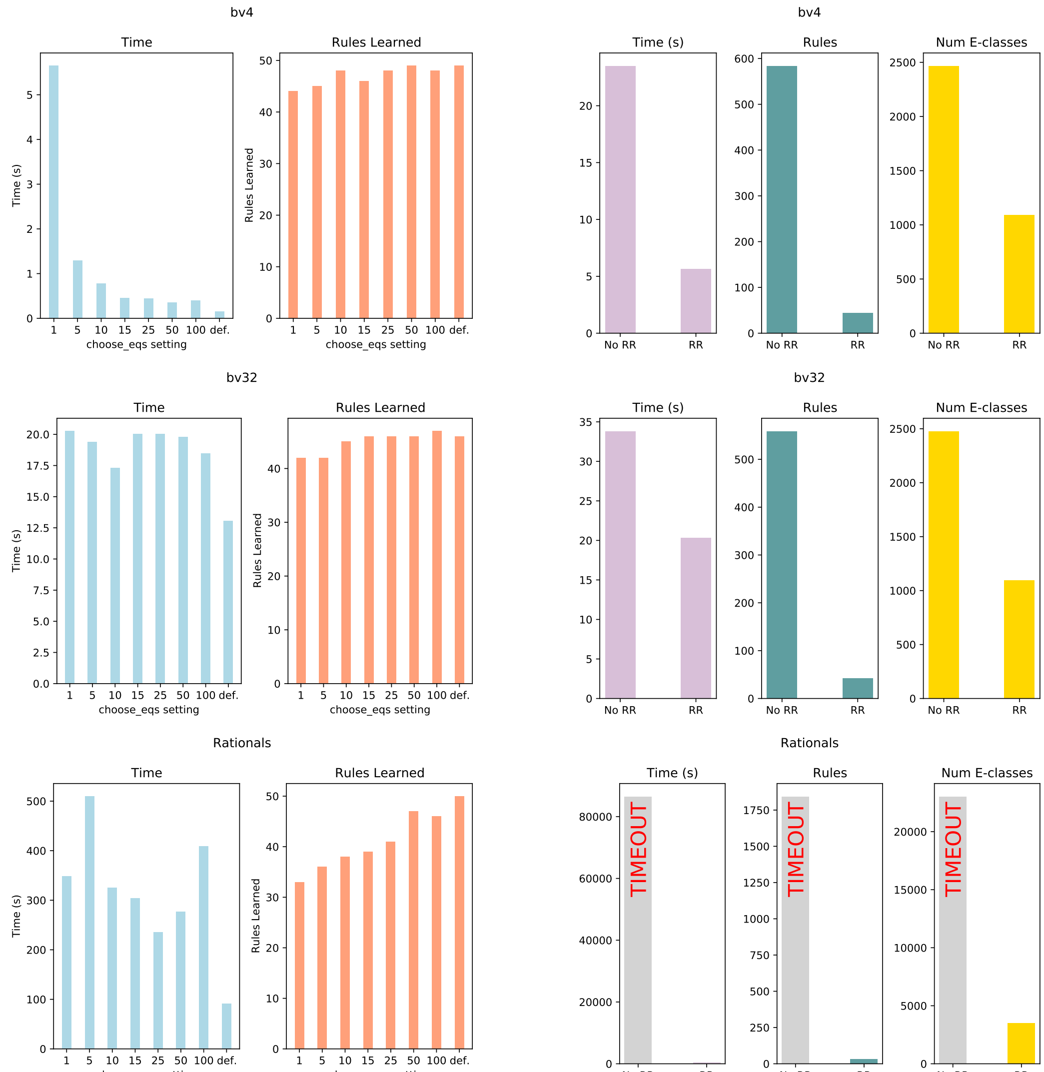
As a rewrite rule inference *framework*, Ruler provides parameters that can be varied to support different user-specified domains. In particular, the "learning rate" parameter n for choose_eqs and the choice of validation method present potential tradeoffs in terms of synthesis time, ruleset quality, and soundness.

In this section we evaluate how varying these parameters affects three representative domain categories: (1) small domains like bitvector-4 where exhaustive model checking by complete cvecs is feasible, (2) large domains like bitvector-32 with non-uniform behavior that typically require constraint solving for validation, and (3) infinite domains like rationals with uniform behavior where fuzzing may be sufficient for validation. We additionally profile Ruler's search to see the impact of run_rewrites and compare cvec generation strategies.

6.1 Profiling Ruler Search, Varying `choose_eqs`, and Ablating `run_rewrites`

To help guide our study of Ruler's search, we first profiled how much time Ruler spent in each phase across our representative domains. Figure 8 plots the results using model checking for bitvector-4 and SMT for both bitvector-32 and rationals. run_rewrites (Figure 4) shows time compacting the term space with learned rules, rule_discovery (cvec_match, Figure 4) shows time discovering candidate rules, rule_minimization (choose_eqs, Figure 5) shows time selecting and minimizing rules, and rule_validation (is_valid, Figure 5) shows time validating rules. We ran each experiment in this section 10 times, for two iterations of Ruler, and plot mean values.[10]

[10]For rationals, Ruler iterations correspond to expression depth, while for bitvector-4 and bitvector-32 it corresponds to number of connectives. Relative standard deviation across all experiments was always below 0.033.



(a) Comparison of choose_eqs across values of n. def corresponds to $n = \infty$, Ruler's default configuration. Larger n values are generally faster but produce slightly more rules.

(b) For choose_eqs with $n = 1$, run_rewrites is essential for both speed and synthesizing small rulesets. "No RR" is Ruler without run_rewrites. For rationals, with "No RR" did not complete in 24 hours; with RR, it completed in 350 seconds.

Fig. 9. Comparison of Ruler's performance across variations of the search algorithm.

After validation (for applicable domains), Ruler spends most of its search time minimizing candidate rules. This is expected because choose_eqs minimizes the set of candidates C by invoking equality saturation and $|C|$ can reach roughly 10^6. Ruler uses a "learning rate" parameter n in choose_eqs to control how aggressively it tries to minimize rules (line 27, Figure 5). When $n = 1$, Ruler selects only a single "best" rule, requiring more rounds of selecting and shrinking candidate

cvec		random: 0		random: 10		random: 100		random: 1000		SMT	
4-bit Bitvector											
C	343	—	—	49/2	0.1s	49/-	0.1s	49/-	0.1s	49/-	1.1s
C	1331	49/-	0.1s	49/-	0.1s	49/-	0.1s	49/-	0.1s	49/-	1.0s
C	4096	49/-	0.2s	49/-	0.2s	49/-	0.2s	49/-	0.2s	49/-	1.1s
R	343	49/-	0.1s	49/-	0.1s	49/-	0.1s	49/-	0.1s	49/-	1.0s
R	1331	49/-	0.1s	49/-	0.1s	49/-	0.1s	49/-	0.1s	49/-	1.0s
R	4096	49/-	0.2s	49/-	0.2s	49/-	0.2s	49/-	0.2s	49/-	1.1s
32-bit Bitvector											
C	343	—	—	—	—	—	—	—	—	46/-	13.6s
C	1331	46/-	0.1s	46/-	0.1s	46/-	0.1s	46/-	0.1s	47/-	13.2s
C	6859	46/-	0.2s	46/-	0.2s	46/-	0.2s	46/-	0.2s	47/-	13.3s
R	343	—	—	—	—	—	—	—	—	37/-	7.6s
R	1331	—	—	—	—	—	—	—	—	37/-	7.7s
R	6859	—	—	—	—	—	—	—	—	37/-	7.8s
Rational									cell format: #sound/#unsound		time
C	27	—	—	50/-	16.8s	50/-	20.6s	47/-	55.3s	50/-	122.4s
C	125	46/-	33.3s	46/-	32.7s	46/-	32.8s	46/-	34.3s	46/-	38.2s
C	729	52/-	342.3s	52/-	341.6s	52/-	357.2s	49/-	341.3s	52/-	349.5s
R	27	50/-	16.4s	50/-	16.5s	46/-	16.0s	48/-	17.4s	50/-	19.5s
R	125	49/-	64.0s	49/-	65.2s	48/-	65.1s	48/-	64.2s	49/-	68.5s
R	729	50/-	413.6s	49/-	311.5s	49/-	308.4s	50/-	414.0s	49/-	316.7s

Table 2. Comparing cvec generation and validation strategies. First column shows cvec generation approach and length: C = Cartesian product of hand-picked values, R = randomly sampled values. Middle columns correspond to validation by random testing over varying number of samples, and the last column is for SMT based validation. We check the rules for soundness with a separate, SMT-based post-pass and report the number of sound/unsound rules and the synthesis time in seconds. A dashed cell indicates that Ruler detected unsoundness and crashed.

rules. By default, $n = \infty$, which causes Ruler to select a minimized version of that iteration's entire candidate ruleset.

Figure 9a shows how varying n affects overall search time and the resulting ruleset size: more aggressive minimization at $n = 1$ is slower but produces smaller rulesets relative to the default $n = \infty$. This is expected: since the set of candidate rules C is typically small compared to the entire term e-graph T, it is more efficient to iteratively shrink C than to repeatedly shrink T with only a few additional rules each time. These rulesets all generally had equivalent inter-derivability (Section 4), with the minimum ratio of 0.92 due to heuristics.

To understand how much shrinking the term e-graph T impacts search, we also compared running Ruler with and without run_rewrites. We set $n = 1$ for minimization as that setting relies the most on run_rewrites.[11] As Figure 9b shows, run_rewrites significantly improves search time and ruleset size while simultaneously requiring less space to store T.

[11] We also conducted this experiment with the default $n = \infty$ and found less pronounced results as expected.

6.2 Sensitivity Analysis for Validation Methods

Figure 8 shows that Ruler spends most of its search time in rule validation when using SMT. To investigate the relative performance and soundness of other validation methods, we compared various strategies for constructing cvecs and applying increasing levels of fuzzing during rule synthesis across our representative domains.

Table 2 shows that fuzzing can be used to synthesize surprisingly sound rulesets, with only a single configuration (bitvector-4, C = 343, random = 10) producing any unsound rules. This is because equality saturation tends to "amplify" the unsoundness of invalid rules. Similar to inadvertently proving `False` in an SMT solver, unsound rules in equality saturation quickly lead to attempted merges of distinct constants or e-classes with incompatible cvecs. Ruler detects such unsound merge attempts and exits immediately after reporting an error to the user along with the rule that triggered the bogus merge (which may or may not be the rule ultimately responsible for introducing unsoundness in the e-graph). These "equality saturation soundness" crashes are indicated by "—" entries in Table 2. For the sole configuration that found unsound results without crashing, we reran the experiment with modestly increased resource limits and Ruler was able to detect the unsoundness without SMT. Despite this encouraging result, we emphasize that fuzzing alone cannot guarantee soundness in general.

For small domains like bitvector-4 with 3 variables, Ruler can employ exhaustive cvecs to quickly synthesize small, sound rulesets. For larger domains like bitvector-32, exhaustive cvecs are infeasible: even for 2 variables they would require cvecs of length $(2^{32})^2$. Larger domains like bitvector-32 with subtly non-uniform behavior especially require verification or good input sampling since, e.g., even if $x > 0$ it is possible to have $x * x = 0$.

To mitigate this challenge, Ruler allows cvecs to be randomly sampled (R), or populated by taking the Cartesian product (C) of sets of user-specified "interesting values".For example, the Ruler bitvector domains use values around 0, 1, MIN, MAX, rationals uses 0, 1, 2, -1, -2, $\frac{1}{2}$, ...

We found that for uniform domains like rationals, using small cvecs with some random testing is sufficient for generating sound rules. For rationals, the low variability in the number of rules learned across the different configurations is an artifact of Ruler's minimization heuristics and cvec_matching — grouping e-classes based on cvecs (Section 3.3) to determine which terms are matched to become potential rewrite rule candidates.

For bitvector-32, we found that seeding cvecs with interesting constants was more effective than random cvecs— due to the nonuniform nature of larger bitvectors, naively sampling random cvec values was insufficient for uncovering all the edge cases during rule validation, but when unsound rules were added, Ruler crashed due to "equality saturation soundness" violations.

6.3 Handling Domain Updates

An important potential application of automatic rewrite rule synthesis is in helping programmers explore the design space for rewrite systems in new domains or during maintenance of existing systems to handle updates when a domain's semantics evolves. To simulate such a scenario, we took inspiration from the recent change in Halide's semantics[12] to define $x/0 = 0$, and similarly changed the implementation of division for the rationals domain to make the operator total.

Under the original rationals semantics where division by zero is undefined, Ruler learns 50 rules in roughly 123 seconds with SMT validation and 47 equivalent rules in roughly 21 seconds fuzzing 100 random values for validation (Table 2).

Making division total for the rationals domain in Ruler required changing a single line in the rationals interpreter. After this change, we used fuzzing with 100 random values to synthesize

[12] https://github.com/halide/Halide/pull/4439

47 rules in 18 seconds. Since fuzzing is potentially unsound, we also extended SMT support in our modified version of rationals with an additional 12 line change. We then synthesized 47 rules in 59 seconds using SMT validation and checked that both the new fuzzing-inferred and new SMT-inferred rulesets could each completely derive the other (the rulesets were identical).

Comparing the rulesets between the original and updated division semantics revealed expected differences, e.g., only the original rulesets contained $x/x \leftrightsquigarrow 1$ and only the updated rulesets contained $x/0 \leftrightsquigarrow 0$. Checking derivability between the old and new rulesets identified 5 additional rules that were incompatible between the semantics, shedding additional light on the consequences of the change to division semantics.

7 LIMITATIONS AND FUTURE WORK

Like any synthesis tool, Ruler uses limits, caps, and heuristics to achieve practical performance. Of particular note are the heuristics in choose_eqs — select for scoring candidate rules, and the *step* size used to determine the number of rules to process at a time. select uses syntactic, size-based heuristics to approximate richer concepts like subsumption. As Section 6 showed, these heuristics can affect the size of the ruleset, though not significantly.

While Ruler's equality saturation approach eliminates α-equivalent rules, it does not eliminate α-equivalent terms from the enumeration set T. Doing so could significantly reduce the enumeration space and increase performance.

Ruler admits other implementations of term enumeration via add_terms (Section 3), but our default prototype implementation only explores complete enumeration. Stochastic enumeration, potentially based on characteristics of a workload, could further improve scalability. Ruler could also seed the initial term e-graph used for enumeration with expressions drawn from interesting workloads, e.g., benchmark suites or traces from users. This seeding could both speed up rule inference and improve the effectiveness of generated rules. Enumeration could also be limited to a semantically meaningful language subset; for example, it is possible to learn a subset of rules over reals by learning rules over rationals — the latter should be faster (as rationals have fewer operators), and the rational rules learned should remain sound when lifted to operating over reals.

Ruler already provides limited support for partial operators like div by allowing the interpreter to return a null value (Section 3). Ruler does not, however, infer the *conditions* that ensure that partial operators succeed. Furthermore, some rewrites are total but still depend on some condition being met: for example $|x| \leftrightsquigarrow x$ only when x is non-negative. An extension to Ruler could possibly to infer such side conditions based on "near cvec matches" where only a few entries differ between two e-classes's cvecs, building on prior work [Menendez and Nagarakatte 2017] that infers preconditions for peephole optimizations.

8 RELATED WORK

This section discusses prior work on rewrite rule synthesis. Most of the work in this area focuses on domain-specific rewrite synthesis tools, unlike Ruler, which is a domain-general framework for synthesizing rules, given a grammar and interpreter. We also focus on other framework-based approaches for rule synthesis and compare Ruler against them.

8.1 Rule synthesis for SMT solvers

Pre-processing for SMT solvers and related tools often involves term rewriting. Past work has attempted to automatically generate rules for such rewrites. Recent work from Nötzli et al. [2019] is the most relevant to Ruler. They present a partially-automated approach for enumerating rewrite rules for SMT solvers. We provide a comparison with Ruler in Section 4. The main commonality between our work and theirs is the use of sampling to detect new equivalences; this is similar to cvec matching in Ruler. Ruler's approach is unique in its use of equality saturation to shrink both the candidate rules and the set of enumerated terms. Nötzli et al. [2019] apply filtering strategies like subsumption, canonical variable ordering, and semantic equivalence; their term enumeration is based on Syntax-Guided Synthesis (SyGuS). Their tool can be configured to use an initial set of rules from cvc4 [Barrett et al. 2011] to help guide their search for new rules. Ruler currently synthesizes generalized rules from scratch. Like Nötzli et al. [2019], Ruler generates rules that do not guarantee a reduction order, since it synthesizes rules like commutativity and associativity. To mitigate exponential blowups when using such rules, one approach is limiting the application of these rules [Willsey 2021]. Prior work [Nandi et al. 2020] has also demonstrated the use of inverse transformations to mitigate the AC-matching problem. Newcomb et al. [2020] have recently explored how to infer termination orders for non-terminating rules, which could be interesting future work for Ruler as well.

SWAPPER [Singh and Solar-Lezama 2016] is a tool for automatically generating formula simplifiers using machine learning and constraint-based synthesis. SWAPPER finds candidate patterns for rules by applying machine learning on a corpus of formulae. Conditions in SWAPPER are inferred by first enumerating all possible expressions from a predicate language. Then, the right hand side of the rule is synthesized by fixing the predicate and using Sketch [Solar-Lezama 2008]. Romano and Engler [2013] infer reduction rules to simplify expressions before passing them to a solver, thereby reducing the number of queries sent to, and subsequently time spent, in the solver. The rules are generated by symbolic program evaluation, and validated using a theorem prover. Nadel [2014] used a combination of constant propagation and equivalence propagation to speed up bit-vector rewriting in various solvers. Several other papers [Hansen 2012; Niemetz et al. 2018] propose algorithms and tools for automatically generating rules for bit-vectors. Newcomb et al. [2020] recently used program synthesis and formal verification to improve the rules in Halide. They focus only on integer rules and rely on mining specific workloads to identify candidates for rewrites. Preliminary experiments in Ruler indicate that supporting integers and even floats is achievable with random sampling or other validation approaches.

8.2 Instruction Selection and Graph Substitutions

Several tools have been proposed to automatically synthesize rewrite rules for instruction selectors. Buchwald et al. [2018] propose a hybrid approach called "iterative CEGIS", combining enumeration with counter-example guided inductive synthesis (CEGIS) to speed up the synthesis of a rule library. As the authors describe in the paper, their tool does not support division, and they also do not infer any rules over floats, since the SMT solvers they rely on are not suitable for these domains. Ruler can be used to infer rules for domains not supported by SMT or that have different semantics because its core algorithm does not rely on SMT — it uses SMT for verifying rules for domains that are supported, but Section 6 shows that it is straightforward to use other validation techniques, or even change the semantics of the language and get a new set of rewrites. Dias and Ramsey [2010] proposed a heuristic search technique for automatic generation of instruction selectors given a machine description. Their work uses algebraic laws to rewrite expressions to expand the space of expressions computable in a machine.

TASO [Jia et al. 2019] is a recent tool that automatically infers graph substitutions for optimizing graph-based deep learning computations. TASO automatically generates rules and verifies them using Z3 [De Moura and Bjørner 2008]. To generate the candidates, TASO enumerates expressions from a grammar up to a certain depth and applies random testing to find equivalences, similar to Ruler. To verify the rules, TASO uses a set of axioms that express operator properties in first order logic. The axioms are used to prove that the generated rules for graph substitution are correct. TASO uses subsumption to eliminate rules that are direct special cases of other rules.

8.3 Theory Exploration

QuickSpec [Claessen et al. 2010] is a tool that automatically infers specifications for Haskell programs from tests in the form of algebraic equations. Their approach is similar to Ruler in the sense that they too use tests to find potential equivalences between enumerated terms and filter out equations that are derivable from others.

Equations generated by QuickSpec have been used in an inductive theorem prover called HipSpec [Claessen et al. 2013] to prove other properties about Haskell programs and also integrated with Isabelle/HOL [Johansson et al. 2014]. TheSy [Singher and Itzhaky 2021] uses a symbolic equivalence technique for theory exploration to generate valid axioms for algebraic data types (ADTs). TheSy also uses e-graphs (specifically the egg library) to find equivalences and filter out redundant axioms via term rewriting. Compared to other tools [Johansson et al. 2014], TheSy typically found fewer, more powerful axioms. Using Ruler for theory exploration [Johansson et al. 2010], especially for ADTs would be an interesting experiment in the future.

8.4 Peephole Optimizations

The Denali [Joshi et al. 2002] superoptimizer first showed how to use e-graphs for optimizing programs by applying rewrite rules. Tate et al. [2009] first introduced equality saturation, generalizing some of the ideas in Denali to optimize programs with complex constructs like loops, and conditionals. Since then, multiple tools have used and further generalized equality saturation as a technique for program synthesis, optimization, and verification [Nandi et al. 2020; Panchekha et al. 2015; Premtoon et al. 2020; Stepp et al. 2011; Wang et al. 2020; Wu et al. 2019]. All these tools rely on the implicit assumption that the rewrite rules will be provided to the tool. These rulesets are typically written by a programmer and therefore can have errors or may not be complete. Several tools have automated peephole optimization generation [Bansal and Aiken 2006; Davidson and Fraser 2004; Menendez and Nagarakatte 2017]. Bansal and Aiken [2006] presented a tool for automatically inferring peephole optimizations using superoptimization, using exhaustive enumeration to generate terms up to a certain depth, and leveraging canonicalization to reduce the search space. They use fingerprints to detect equivalences by grouping possibly equivalent terms together based on their evaluation on a few assignments. Grouping likely equivalent terms can eliminate many invalid candidate rules from even being generated in the first place. Ruler's use of cvecs is similar to the idea of fingerprints.

Several other papers [Schkufza et al. 2014; Sharma et al. 2015] have extended and/or use STOKE for synthesizing superoptimizations. Alive-Infer [Menendez and Nagarakatte 2017] is a tool for automatically generating pre-conditions for peephole optimizations for LLVM. Alive-Infer works in three stages: first, it generates positive and negative examples whose validity is checked using an SMT solver. It then uses a predicate enumeration technique to learn predicates, which are used as preconditions. Finally, it uses a boolean formula learner to generate a precondition. Menendez et al. [2016] also developed Alive-FP, a tool that automatically verifies peephole optimizations involving floating point computations. Recently, Lopes et al. [2021] published Alive2 which provides bounded, fully automatic translation validation, while handling undefined behaviour.

9 CONCLUSION

This paper presented a new technique for automatic rewrite rule inference using equality saturation. We identified three key steps in rule inference and proposed Ruler, an equality saturation-based framework that can be used to infer rule-based optimizations for diverse domains.

Ruler's key insight is that equality saturation makes each of the three steps of rule inference more efficient. We implemented rule synthesis in Ruler for booleans, bitvectors, and rationals. We compared Ruler against a state-of-the-art rule inference tool in CVC4; Ruler generates significantly smaller rulesets much faster. We presented a case study showing how Ruler infers rules for complex domains like rationals. Our end-to-end results show that Ruler-synthesized rules can replace and even surpass those generated by domain experts over several years.

We hope that this work energizes the community around equality saturation and incites further exciting research into equality saturation for rewrite rule synthesis.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their thoughtful feedback. We are grateful to Andrew Reynolds for answering our questions about CVC4's rule synthesis component, to Pavel Panchekha and Oliver Flatt for helping us run Herbie with Ruler's rules, and to Sorawee Porncharoenwase and Jacob Van Geffen for helping with Rosette. Thanks to Steven S. Lyubomirsky and Bill Zorn for allowing us to run experiments on their research servers. Thanks to Talia Ringer, Martin Kellogg, Ben Kushigian, Gus Henry Smith, and Mike He for their feedback on earlier drafts of the paper.

REFERENCES

Franz Baader and Tobias Nipkow. 1998. *Term Rewriting and All That.* Cambridge University Press. https://doi.org/10.1017/CBO9781139172752

Leo Bachmair, IV Ramakrishnan, Ashish Tiwari, and Laurent Vigneron. 2000. Congruence closure modulo associativity and commutativity. In *International Workshop on Frontiers of Combining Systems.* Springer, 245–259.

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

Hendrik Pieter Barendregt, Marko CJD van Eekelen, John RW Glauert, J Richard Kennaway, Marinus J Plasmeijer, and M Ronan Sleep. 1987. Term graph rewriting. In *International conference on parallel architectures and languages Europe.* Springer, 141–158.

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.

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`.

Yves Bertot and Pierre Castran. 2010. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions* (1st ed.). Springer Publishing Company, Incorporated.

M. Bezem, J.W. Klop, E. Barendsen, R. de Vrijer, and Terese. 2003. *Term Rewriting Systems.* Cambridge University Press. https://books.google.ca/books?id=oe3QKzhFEBAC

Gabriel Hjort Blindell. 2013. Survey on Instruction Selection: An Extensive and Modern Literature Review. *CoRR* abs/1306.4898 (2013). arXiv:1306.4898 http://arxiv.org/abs/1306.4898

Peter Borovanskỳ, Claude Kirchner, Hélene Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. 1998. An overview of ELAN. *Electronic Notes in Theoretical Computer Science* 15 (1998), 55–70.

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

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18).* USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

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.

Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. 2007. *All About Maude-A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting*

Logic. Vol. 4350. Springer.

Jack W. Davidson and Christopher W. Fraser. 2004. Automatic Generation of Peephole Optimizations. *SIGPLAN Not.* 39, 4 (April 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

Nachum Dershowitz. 1982. Orderings for term-rewriting systems. *Theoretical computer science* 17, 3 (1982), 279–301.

Nachum Dershowitz. 1987. Termination of rewriting. *Journal of symbolic computation* 3, 1-2 (1987), 69–115.

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

João Dias and Norman Ramsey. 2010. Automatically Generating Instruction Selectors Using Declarative Machine Descriptions. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. Association for Computing Machinery, New York, NY, USA, 403–416. https://doi.org/10.1145/1706299.1706346

Steven Eker. 2003. Associative-commutative rewriting on large terms. In *International Conference on Rewriting Techniques and Applications*. Springer, 14–29.

A. P. Ershov. 1958. On Programming of Arithmetic Operations. *Commun. ACM* 1, 8 (Aug. 1958), 3–6. https://doi.org/10.1145/368892.368907

Trevor Alexander Hansen. 2012. *A constraint solver and its application to machine code test generation.* Ph.D. Dissertation. Melbourne, Australia. Advisor(s) Schachte, Peter and Søndergaard, Harald.

Herbie. 2021a. Herbie can generate more-complex expressions that aren't more precise. https://github.com/uwplse/herbie/issues/261#issuecomment-680896733.

Herbie. 2021b. Optimize floating-point expressions for accuracy. https://github.com/uwplse/herbie/issues.

James C. Hoe and Arvind. 2000. *Hardware Synthesis from Term Rewriting Systems.* Springer US, Boston, MA, 595–619. https://doi.org/10.1007/978-0-387-35498-9_52

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

Hélène Kirchner. 2015. Rewriting strategies and strategic rewrite programs. In *Logic, Rewriting, and Concurrency*. Springer, 380–403.

Donald E Knuth and Peter B Bendix. 1983. Simple word problems in universal algebras. In *Automation of Reasoning*. Springer, 342–376.

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Arnaud Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *CGO 2021*.

Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: Bounded Translation Validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 65–79. https://doi.org/10.1145/3453483.3454030

Salvador Lucas. 2001. Termination of rewriting with strategy annotations. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 669–684.

Henry Massalin. 1987. Superoptimizer: A Look at the Smallest Program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems* (Palo Alto, California, USA) *(ASPLOS II)*. IEEE Computer Society Press, Washington, DC, USA, 122–126. https://doi.org/10.1145/36206.36194

W. M. McKeeman. 1965. Peephole Optimization. *Commun. ACM* 8, 7 (July 1965), 443–444. https://doi.org/10.1145/364995.365000

David Menendez and Santosh Nagarakatte. 2017. Alive-Infer: Data-Driven Precondition Inference for Peephole Optimizations in LLVM. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*

(Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 49–63. https://doi.org/10.1145/3062341.3062372

David Menendez, Santosh Nagarakatte, and Aarti Gupta. 2016. Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9837)*, Xavier Rival (Ed.). Springer, 317–337. https://doi.org/10.1007/978-3-662-53413-7_16

Alexander Nadel. 2014. Bit-Vector Rewriting with Automatic Rule Generation. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. Springer-Verlag, Berlin, Heidelberg, 663–679. https://doi.org/10.1007/978-3-319-08867-9_44

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

Charles Gregory Nelson. 1980. *Techniques for Program Verification.* Ph.D. Dissertation. 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).

Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark Barrett, and Cesare Tinelli. 2018. Solving Quantified Bit-Vectors Using Invertibility Conditions. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 236–255.

Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. 2002. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* Springer-Verlag, Berlin, Heidelberg.

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

Doron Peled. 1998. Ten years of partial order reduction. In *International Conference on Computer Aided Verification*. Springer, 17–28.

Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. 2001. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *2001 Haskell Workshop* (2001 haskell workshop ed.). ACM SIGPLAN. https://www.microsoft.com/en-us/research/publication/playing-by-the-rules-rewriting-as-a-practical-optimisation-technique-in-ghc/

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

Racket. 2021. Racket, the Programming Language. https://racket-lang.org/.

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

Anthony Romano and Dawson Engler. 2013. Expression Reduction from Programs in a Symbolic Binary Executor. In *Model Checking Software*, Ezio Bartocci and C. R. Ramakrishnan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 301–319.

Rust. 2021a. Rust BigInt Library. https://docs.rs/num-bigint/0.4.0/num_bigint/struct.BigInt.html.

Rust. 2021b. Rust Rational Library. https://docs.rs/num-rational/0.4.0/num_rational/struct.Ratio.html.

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic Optimization of Floating-Point Programs with Tunable Precision. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 53–64. https://doi.org/10.1145/2594291.2594302

Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2015. Conditionally Correct Superoptimization. *SIGPLAN Not.* 50, 10 (Oct. 2015), 147–162. https://doi.org/10.1145/2858965.2814278

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.

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.

Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph.D. Dissertation. Berkeley, CA, USA. Advisor(s) Bodik, Rastislav. AAI3353225.

Michael Stepp, Ross Tate, and Sorin Lerner. 2011. Equality-Based Translation Validator for LLVM. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 737–742.

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

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

Eelco Visser. 2001a. Stratego: A language for program transformation based on rewriting strategies system description of stratego 0.5. In *International Conference on Rewriting Techniques and Applications*. Springer, 357–361.

Eelco Visser. 2001b. A survey of rewriting strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science* 57, 2 (2001).

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).

Deborah L. Whitfield and Mary Lou Soffa. 1997. An Approach for Exploring Code Improving Transformations. *ACM Trans. Program. Lang. Syst.* 19, 6 (Nov. 1997), 1053–1084. https://doi.org/10.1145/267959.267960

Max Willsey. 2021. egg Documentation. https://docs.rs/egg/0.6.0/egg/struct.BackoffScheduler.html.

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.

Chenming Wu, Haisen Zhao, Chandrakana Nandi, Jeffrey I. Lipton, Zachary Tatlock, and Adriana Schulz. 2019. Carpentry Compiler. *ACM Transactions on Graphics* 38, 6 (2019), Article No. 195. presented at SIGGRAPH Asia 2019.

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