What is a "Good" Ruleset? Comparing Rulesets Using Equality Saturation

CYNTHIA RICHEY, University of Washington, USA

1 PROBLEM & MOTIVATION

Rewrite rules are critical in equality saturation, a technique with applications in compiler optimization, formal verification, and program synthesis. However, the process of maintaining and updating rulesets is difficult, in no small part because a *valid* ruleset is not necessarily a *good* ruleset. If a ruleset is too small, then results will be suboptimal due to missing rules; on the other hand, rules that are not *useful*, either because they are not used by the application or because they are redundant, will cause equality saturation to hit its resource limits prematurely. Therefore, it is important for a ruleset to contain a small, effective set of axioms. An intuitive method for comparing rulesets is needed. We propose the use of the *derivability* heuristic, where a rule is said to be *derivable* by a ruleset, R, if its left- and right-hand sides are determined to be equivalent using R's rules. Here, we expand the notion of derivability introduced in prior work, argue that it should be defined in relation to the equality saturation application in which it is used, and evaluate different definitions of derivability on a variety of domains.

2 BACKGROUND ON E-GRAPHS & EQUALITY SATURATION

The e-graph data structure represents an equivalence relation over terms [1, 3]. It is comprised of a set of *e-classes*, which each contain a set of equivalent e-nodes. Each e-node is a function $f(c_1, c_2, \ldots)$ with children e-classes c_i. An e-graph is said to *represent* a term t if any of its e-classes represent t; an e-class represents t if any e-node in the class represents it. An e-node $f(c_1, \ldots, c_n)$ represents a term $f(t_1, \ldots, t_n)$ if each c_i represents t_i.

A rewrite rule, $l \rightsquigarrow r$, states that a pattern l can be rewritten as r. Equality saturation [4, 5] is an alternative to traditional term rewriting systems that uses an e-graph to apply rewrites: instead of *replacing* subterms of a term t, they are merged into the same e-class. Since it is non-destructive, rewrites can be applied in any order until the e-graph *saturates*—that is, no new e-nodes are added to the e-graph following rule application. In practice, however, saturation is rare: iteration or node limits are usually necessary to constrain the e-graph. Figure 1 presents the core equality saturation algorithm, as well as an example of applying a rule. Equality saturation can be used in many applications, including optimization and equivalence-checking. In optimization, a term t is transformed into an equivalent term t', which is better according to some heuristic, whereas equivalence-checking determines whether two given terms are semantically equivalent.

3 PRIOR WORK

The derivability metric was originally introduced by Nandi et al. in Ruler[2], an equality saturation tool for automatically synthesizing rewrite rules. Rule synthesis is a technique that uses automated theorem proving to synthesize rewrite rules for a user-specified domain. A rule ($l \rightsquigarrow r$) is considered *derivable* by a ruleset R if running equality saturation using R on an e-graph initialized with both l and r causes the two e-classes to merge. A ruleset's *deriving ratio* with respect to another ruleset is the proportion of that ruleset it can derive—so, if R_1 has a deriving ratio of 1 with respect to R_2, but R_2 has a deriving ratio of $2:1$ when deriving R_1, $R1$ would be considered more "powerful".

Derivability is used in two ways in the Ruler tool. First, rule synthesis generates a ruleset containing many redundant rules (the "candidate ruleset"). A minimal subset of rules is found by iteratively choosing rules from the candidate ruleset and eliminating candidates that are derivable

```
1   def equality_saturation(t, R):
2     egraph = empty_egraph()
3     c_root = egraph.add(t)
4     saturated = False
5     while (not saturated) and (not timeout()):
6       saturated = True
7       for ℓ ⇝ r in R:
8         for (σ, c_ℓ) in egraph.search(ℓ):
9           c_r = egraph.add(σ(r))
10          if not egraph.same_eclass(c_ℓ, c_r):
11            egraph.unions(c_ℓ, c_r)
12            saturated = False
13    return egraph.extract_best(c_root)
```

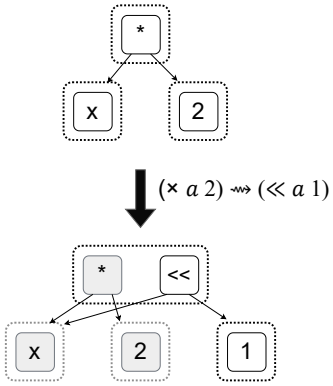

Fig. 1. (Left) The core equality saturation algorithm, which takes a term t and a ruleset R and outputs a term t' equivalent to t [2, 4, 5]. (Right) Applying the rewrite $(\times\ a\ 2) \rightsquigarrow (\ll\ a\ 1)$. Dotted boxes represent e-classes; solid boxes represent e-nodes. The lower e-graph shows that the e-classes for the terms $(\times\ a\ 2)$ and $(\ll\ a\ 1)$ have merged following the application of the rule. Examples are inspired by prior work [2, 5].

from these rules. Second, derivability is used to evaluate the quality of resulting rulesets—e.g., comparing a synthesized boolean ruleset to those from standard textbooks.

4 APPROACH & UNIQUENESS

Prior work narrowly defines a "quality" ruleset as a ruleset with more proving power. In practice, users may want to tailor the definition of derivability to the needs of their equality saturation application. We identify two potential applications: optimization, and equivalence checking. We advocate for defining derivability differently depending on the application.

It is possible to implement different versions of derivability by making simple changes to the initial context of the e-graph. We introduce a stricter definition—termed `lhs`—where, given a ruleset R and a rule $l \rightsquigarrow r$, the e-graph is seeded with only the left-hand side of the rule. If, following equality saturation, r is in the same e-class as l, then the rule is considered derivable. We propose this definition for use in the *optimization* application. In contrast, Ruler used a definition of derivability—termed `lhs-rhs`—which we find is suited to *equivalence-checking* applications.

To illustrate the difference, consider the rule $a \rightsquigarrow b$ and a ruleset containing the rule $b \rightsquigarrow a$. In an e-graph seeded with both a and b, $b \rightsquigarrow a$ will merge a with b; however, when only a is present in the e-graph, the rule $b \rightsquigarrow a$ will never run. In other words, under `lhs` derivability, $a \rightsquigarrow b$ cannot derive $b \rightsquigarrow a$, whereas, under `lhs-rhs` derivability, it can.

Even in cases where a rule is theoretically derivable using both definitions, the `lhs` definition is stricter. For example, given the rule $x \rightsquigarrow z$ and the ruleset $\{x \rightsquigarrow y, y \rightsquigarrow z, z \rightsquigarrow y\}$, the e-classes of x and z will eventually merge using either `lhs` or `lhs-rhs`. However, in the `lhs` case, it will take an additional iteration of equality saturation. In general, given k iterations, the `lhs-rhs` definition requires that l and r merge within $2k$ rule applications, as opposed to k in the `lhs` case. This is because every term in the e-graph is a target for rewrites at each iteration, so more terms enable longer chains of transformations.

Prior work assumes that the goal of derivability is to establish that there *exists* some chain of rewrites in R that can merge l and r, but this may not always be the case. When *optimizing* expressions, direct rules may be preferred to indirect ones. Say we have the rule $r : a \rightsquigarrow c$ and the rulesets $R_1 : \{a \rightsquigarrow c\}$ and $R_2 : \{a \rightsquigarrow b, c \rightsquigarrow b\}$. Under `lhs-rhs`, both rulesets derive r in the same

number of iterations—but using `lhs`, only R_1 can derive r. In the *equivalence-checking* scenario, the "better" ruleset is the one that gives us more information (in this case, the involvement of b); in the optimization application, we prefer the ruleset that proves the rule directly.

We also observe that prior work in rule synthesis[2] uses not one but *two* distinct versions of derivability. When rulesets are compared for evaluation purposes, the `lhs-rhs` approach is used; however, during rule minimization (a heuristic used to scale synthesis), the approach is different. After synthesizing a set of valid rules C, which may contain significant redundancy, Ruler uses a heuristic to select a subset of rules, S, from this candidate set. All the terms in the candidate set—that is, the left- and right-hand sides of all the rules—are added to an e-graph, and equality saturation is run using S. Following this process, C is pruned to contain only rules that have not merged following equality saturation using S—that is, it now contains the rules that S cannot *derive*! Significantly looser than even `lhs-rhs`, this definition enables many merges to occur at each iteration of equality saturation, which is useful when working with large e-graphs.

5 RESULTS & CONTRIBUTION

We performed two experiments to compare the different variants of derivability and understand how the different variants interact with equality saturation resource limits, such as iterations and nodes. For the first experiment, we evaluated both derivability metrics, `lhs` and `lhs-rhs`, on increasingly complex domains from Ruler[2]: booleans, bitvectors, and rational numbers. We took the synthesized rulesets from both the main branch [1] of the most recently-updated version of Ruler ("new" rulesets), and the originally-published version ("old" rulesets) [2]. We used the new rulesets to derive the old rulesets, and vice versa, using both derivability metrics. As expected, `lhs` consistently reported a more conservative result than `lhs-rhs`, although the differential varied by domain. Full results can be seen in Figure 2.

Domain	New → Old (`lhs`, `lhs-rhs`)	Old → New
bool	100%, 100%	87.5%, 96.9%
bv4	100%, 100%	38.3%, 41.1%
bv32	100%, 100%	58.3%, 60.0%
rational	97.3%, 100%	52.0%, 58.5%

Fig. 2. Results of using both the `lhs` and `lhs-rhs` derivability metrics to compare rulesets in various domains from the state-of-the-art rule synthesis tool, Ruler [2]. "New" refers to the ruleset obtained from the main branch Ruler and "Old" refers to the ruleset obtained from the originally published version of Ruler.

For the second experiment, we altered Ruler's minimization algorithm to use the `lhs-rhs` version of derivability and compared against Ruler's original algorithm, keeping resource limits the same for both. We used one small domain (boolean) and one large domain (rational). For booleans, the altered version of Ruler found exactly the same rules as the original version; on the rational domain, it timed out before finding any rules, whereas the original algorithm found 131 rules. This suggests that different definitions of derivability suit different contexts and have different resource requirements—here, the original, looser definition of derivability scaled better on larger domains.

In summary, the notion of *derivability* is a valuable heuristic for evaluating ruleset quality in equality saturation applications. We argue that derivability is best viewed as a *suite* of approaches. We explore and characterize the definition of derivability, and identify three algorithms for computing derivability. We presented concrete use-cases and applications for each from prior work and present comparative results. We continue to work in this direction, further evaluating how these variants of derivability interact with resource bounds commonly used in equality saturation applications.

[1] https://github.com/uwplse/ruler
[2] https://github.com/uwplse/ruler/tree/oopsla21-aec

REFERENCES

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

[2] Chandrakana Nandi, Max Willsey, Amy Zhu, Yisu Remy Wang, Brett Saiki, Adam Anderson, Adriana Schulz, Dan Grossman, and Zachary Tatlock. 2021. Rewrite Rule Inference Using Equality Saturation. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 119 (oct 2021), 28 pages. https://doi.org/10.1145/3485496

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

[4] 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

[5] 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.