Saarland University
Faculty of Mathematics and Computer Science
Department of Computer Science

Bachelor's Thesis

Knuth-Bendix Completion for Program Optimization

by
Michael Schifferer

submitted
October 30, 2025

Reviewers:

1. Prof. Dr. Sebastian Hack
2. Prof. Dr. Jan Reineke

Advisor:

Prof. Dr. Sebastian Hack

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne die Beteiligung dritter Personen verfasst habe, und dass ich keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus Veröffentlichungen oder aus anderweitigen fremden Äußerungen entnommen wurden, sind als solche kenntlich gemacht. Insbesondere bestätige ich hiermit, dass ich alle mittels künstlicher Intelligenz betriebenen Software (z.B. ChatGPT) generierten und/oder bearbeiteten Teile der Arbeit kenntlich gemacht und als Hilfsmittel angegeben habe. Mir ist bewusst, dass der Verstoß gegen diese Versicherung zum Nichtbestehen der Prüfung bis hin zum Verlust des Prüfungsanspruchs führen kann.

Declaration of Original Authorship

I hereby declare that this thesis is my own original work and was completed independently, without unauthorized assistance or unacknowledged sources. Any content derived from publications or other external sources, whether quoted verbatim or paraphrased, has been duly acknowledged and clearly marked as such. I hereby confirm that any parts of the thesis produced or modified with the assistance of artificial intelligence-based software (such as ChatGPT) have been clearly indicated as such and that the software used has been listed as a resource. I acknowledge that any breach of this declaration may result in failing the examination and, in severe cases, the right to be examined may be revoked.

Ort/Place, Datum/Date	Unterschrift/Signature

Abstract

Rewrite-based optimization techniques are important tools for program transformation in optimizing compilers. Equality saturation (EqSat) is one such technique that has recently attracted significant research interest. While EqSat has proven to be an effective tool in practice, its feasibility and performance rely heavily on the rule sets used.

Such rule sets are typically handcrafted, which requires in-depth domain knowledge and may miss valuable transformations.

To address this, we evaluate Knuth–Bendix completion (KBC) as a method for automatically generating rewrite rules. The evaluation focuses on the performance of KBC-generated rule sets in EqSat and in a custom greedy rewriting engine.

The evaluation shows that KBC-generated rewrite rules can complement, but not fully replace, handwritten rules in EqSat. When used with greedy rewriting, they enable highly resource-efficient but considerably less effective simplification.

Contents

List of Figures

List of Tables

1 Introduction

The effectiveness of program optimization depends not only on the type of optimizations applied, but also on the order in which they are applied. This is generally known as the *phase ordering problem* and represents a major challenge when designing optimizing compilers.

To solve this problem, *equality saturation* (EqSat) Tate et al. (2011); Willsey et al. (2021b) was introduced. By transforming programs within e-graphs, a special data structure that maintains equivalence classes of programs, EqSat explores the search space of candidate programs systematically, rather than applying transformations sequentially.

While EqSat has proven to be an effective optimization framework in specific applications, its feasibility and performance rely heavily on the rewrite rules used to derive equivalent programs. The rule sets are typically handcrafted to balance the trade-off between enabling a deep search space, to include as many promising candidate programs as possible, and limiting the growth of the e-graph from unproductive rules. Designing such rule sets, therefore, requires strong domain knowledge and intuition, or extensive testing.

Automatically generating rewrite rules that fit the requirements of EqSat could therefore improve the effectiveness of the technique and also enable its use on complex domains. Currently, little is known about the properties of effective rule sets for EqSat, and the available tools and techniques for this purpose are limited.

This work evaluates *Knuth-Bendix completion* (KBC) (Dick, 1991), a technique which has been used successfully in automated theorem proving, as a tool for generating rewrite rules used in program optimization.

To this end, we develop different ways to generate rule sets using KBC based on an initial set, obtained from the test suite of `egg` Willsey et al. (2021b), a popular implementation of EqSat. We also provide a prototype of a rewriting engine specialized for greedy rewriting with KBC-generated rule sets.

We then conduct a series of tests, applying EqSat and greedy rewriting to randomly generated test terms from arithmetic.

The evaluation shows that incorporating KBC-generated rules can improve the effectiveness of EqSat in terms of running time and overall simplification of input terms, compared to the purely handwritten rules. It also reveals that greedy rewriting does not achieve the same level of simplification effectiveness as EqSat under the given conditions, but is much more resource-efficient.

In summary, the contributions of this work are:

- Development of different ways to generate rule sets for optimization using KBC (section 3.1).
- Development of a prototype for greedy rewriting using KBC-generated rule sets (section 3.3.2).
- Experimental evaluation of KBC-generated rule sets to answer the following research questions (chapter 4):
    1. Do KBC-generated rule sets improve equality saturation in terms of simplification effectiveness and running time, compared to handwritten rules?
    2. Do KBC-generated rule sets enable greedy rewriting as a viable alternative to equality saturation in scenarios where minimizing running time and memory usage is essential?

2 Background

To understand the contribution of this work, we first need to establish the three concepts it builds upon: *term rewriting* as a rule-based process of program transformation, *equality saturation* as a rewrite-based technique for program optimization, and *Knuth–Bendix completion* as a procedure for extending sets of rewrite rules. This chapter provides an informal overview of these topics, focusing on the relevant aspects for program optimization.

2.1 Term Rewriting

Term rewriting generally requires two components: a term to be rewritten and a set of directed rewrite rules, also called a *term rewriting system*, or TRS for short. Figure 2.1 serves as an illustration of the rewriting process. Within figure 2.1, we find a set of rewrite rules derived from the basic group axioms for addition, as well as a sequence of rewrites, starting at an arithmetic term t and ending at term *t'*. Note that some axioms are represented by two rules, one for each direction.

Performing a rewrite generally requires matching the left-hand side of one of the rules on some subterm of the term to be rewritten and replacing that subterm with the right-hand side of the rule.

R1: $x + 0 \to x$
R2: $x \to x + 0$
R3: $0 + x \to x$
R4: $x \to 0 + x$
R5: $(x + y) + z \to x + (y + z)$
R6: $x + (y + z) \to (x + y) + z$
R7: $-x + x \to 0$

(a) Set of rewrite rules derived from group axioms for addition.

$$\begin{tikzcd}
\boxed{(- - a + -a) + a} \arrow[d, "\text{R7}"'] \\
\boxed{0 + a} \arrow[d, "\text{R3}"'] \\
\boxed{a}
\end{tikzcd}$$

(b) Derivation of a from $(- - a + -a) + a$.

Figure 2.1: Example: term rewriting process for $(- - a + -a) + a$ using the group axioms for addition.

It is important to note that the semantics of the resulting term *t'* remain the same, since the underlying axioms are equalities. Term rewriting, therefore, allows us to find alternative representations of terms while preserving meaning. The number of possible rewrites and, therefore, the number of derivable terms depends on the rule set. In the example shown in figure 2.1, the rule set contains expansive rules and therefore permits an infinite number of rewrites.

Being able to transform programs into more optimal forms, for some notion of optimality, while preserving meaning, makes term rewriting an interesting approach for program optimization. Crucially, the correctness of the transformations depends solely on the correctness of the employed rewrite rules.

The connection between term rewriting and computer programs becomes even more apparent when examining functional programming. In other words, a functional program may be viewed as a type of deterministic term rewriting system (Baader and Nipkow, 1998).

This connection with functional programming highlights a limitation when applying term rewriting as a tool for program optimization. While pure functional programming does not rely on mutable state and side effects, programming languages utilizing other paradigms do. As a result, term rewriting may only be applied to those parts of programs that behave like pure functions.

Despite this limitation, term rewriting can still be very effective when optimizing programs, for example, to reduce running time. The technique works particularly well in conjunction with other optimizations such as constant propagation and unreachable code elimination. Consider figure 2.2: here, applying the rewrite sequence from figure 2.1 in line two makes it obvious to the compiler that the condition inside the `if`-statement always holds. As a result, the false branch becomes unreachable and can be removed entirely.

Before Optimization

```
1 void example(int a) {
2   int b = ((-(-a) + -a) + a);
3   if ((b - a) == 0) {
4     foo(b);
5   } else {
6     bar(a);
7   }
8 }
```

After Optimization

```
1 void example(int a) {
2   int b = a;
3   foo(b);
4 }
```

Figure 2.2: Example: unreachable code elimination, enabled by rewriting $(- - a + -a) + a$ to a.

2.2 Equality Saturation

Equality Saturation (EqSat) is a technique that uses term rewriting to exhaustively derive equivalent terms for one or more input terms. In essence, EqSat applies all applicable rules at every point in a rewrite sequence. To make exhaustive rewriting feasible, even for TRS that permit an infinite number of derivations, such as in figure 2.1, EqSat uses a special data structure called an e-graph to represent terms compactly.

An e-graph contains e-nodes and e-classes, with the former being term symbols and the latter being equivalence classes that each contain the set of root symbols of equivalent terms. Applying rewrite rules grows the e-graph by adding new nodes, and a rebuilding step ensures that equivalent terms are part of the same e-class.

A major advantage of this data structure is that expansive patterns, such as addition of 0 in arithmetic, don't grow the graph infinitely. As a result, the e-graph may reach a fixpoint, at which additional iterations of rule applications don't change the graph anymore. At this point, saturation is reached, and the optimal term can be extracted from the graph (Willsey et al., 2021b).

Figure 2.3a shows the familiar term $(--a+-a)+a$ as an e-graph. The symbol 'root+' refers to the second addition in the term, but is marked with 'root', so it is easier to identify as the root symbol.

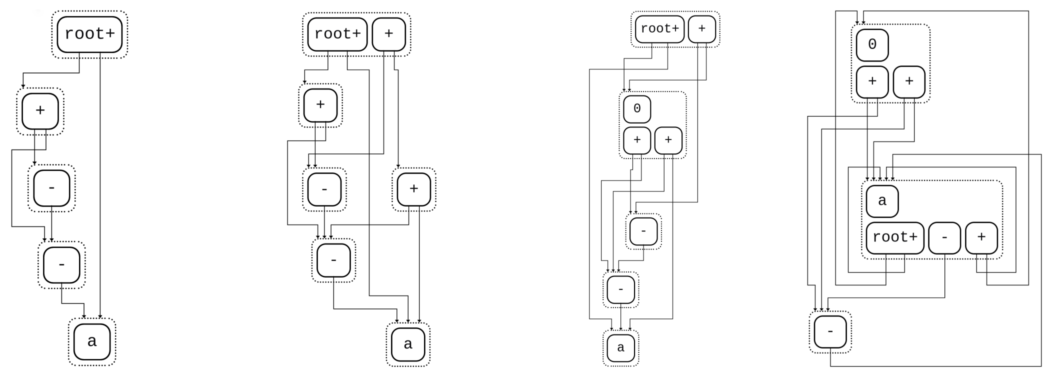
Further, we see that all nodes have their own equivalence classes, indicated by the dashed boxes around them. This changes when we apply associativity as a rule. In figure 2.3b we see that 'root+' now shares an equivalence class with the root symbol of the associative counterpart $(--a+(-a+a))$.

After applying a total of four rules, we end up at figure 2.3d. It is important to note that we applied rules selectively and that more rules could be applied after every step. In practice, this would happen during EqSat; however, the resulting e-graph would quickly become too complex to serve an illustrative purpose.

There are multiple options for terminating EqSat. Ideally, the process ends on saturation of the e-graph. That is, the procedure runs until additional rule applications do not change the e-graph anymore. In this case, EqSat found all possible derivations.

In practice, saturation is uncommon due to EqSat being resource-intensive. Rules such as associativity and commutativity in particular increase graph size exponentially (Zhang and Flatt, 2023), making it likely that for sufficiently complex terms and rule sets, memory becomes a limiting factor. Therefore, in many cases, termination of EqSat depends on specified resource limits.

A key advantage offered by EqSat is that the process of finding terms happens independently of the rewrite goal. After termination, one can extract the desired term by using a cost function. For this, consider again figure 2.3d. By locating the equivalence class containing 'root+', we find the roots of all terms equivalent to



(a) Term $(--a+-a)+a$ (b) After applying $(X+Y)+Z \rightarrow X+(Y+Z)$ (c) After applying $-X + X \rightarrow 0$ (d) After applying $0 + X \rightarrow X$ and $X + 0 \rightarrow X$

Figure 2.3: Example: representation of $(- - a + -a) + a$ as an e-graph and application of four rewrite rules. e-nodes in graph shown as solid boxes, e-classes as dashed boxes.

our original term. The set containing all of these terms consists of all paths from the root nodes to leaves. It is now easy to verify that the rewrite from figure 2.1 contained in the e-graph, by confirming that the node 'a' is inside the e-class of 'root+'. Extracting the optimal term from an e-graph is in itself a challenge and subject of ongoing research (Yin et al., 2025).

2.3 Knuth-Bendix Completion

Knuth-Bendix completion (KBC), unlike EqSat, is a rewrite-based technique that has not seen significant attention in the context of program optimization. Instead, it is primarily used in automated theorem proving. When trying to prove the equivalence of two terms s and t using term rewriting, we need to show that we can either rewrite s to t, t to s, or s and t to a third term u. A KBC-based proof uses the latter approach, although $u = t$ or $u = s$ may hold.

The term u used in such a proof is known as the normal form of the set of equivalent terms it belongs to. If s and t are equal, they must share the same normal form. While this normal form is not explicitly known beforehand, it is uniquely determined as it is the smallest term in its equivalence class according to some ordering. We will refer to this ordering as the *Knuth-Bendix ordering* (KBO). There are a number of formal requirements for an ordering to be considered a KBO, which Baader and Nipkow (1998) elaborate on in detail. Most importantly, KBO is a well-founded simplification order. This implies that for all equivalence classes of ground terms, i.e., terms that do not contain variables, there exists a unique smallest element.

To understand how KBC works in more detail, we will reformulate the example from figure 2.1 as a proof goal. For this let the $t = (- - a + -a) + a$ and $s = a$. In this case, it is obvious that a is the normal form of t. Therefore, one possible proof consists of performing the sequence of rewrites from figure 2.1. The first step for proving this equality using KBC requires replacing the original set of rewrite rules with a set in which, for all rules, the left-hand side is larger under KBO than the right-hand side. This ensures that whenever a rewrite is applied, the term becomes smaller and therefore closer to its normal form.

For the axioms of addition, the directions for all but two axioms coincide with intuition. That is, if a term has fewer symbols, it is smaller. This is reflected by the usage of a weight function in KBO. In the case of associativity and commutativity, both sides of the axiom share the same symbols and therefore their weight is the same. In such a case, KBO orders terms lexicographically. In the case of associativity this means $(X + Y) + Z >_{KBO} X + (Y + Z)$ since $(X + Y) >_{KBO} X$. In the case of commutativity, this method also fails, which is why basic variants of KBC fail when encountering such an axiom. Extensions of the basic KBC algorithm solve this issue by keeping unorderable rules unordered until they are instantiated. Those variants are known as *unfailing* KBC (Dick, 1991) and will be referred to simply as KBC throughout this paper.

After orienting the initial set of axioms, we can already try applying the rules. Figure 2.4 shows the directed rules, as well as a rewrite sequence derived using said rules. Note that the derived sequence differs from figure 2.1 in the first step and as a result does not end with the normal form a, but instead with $- - a$.

R1: $0 + X \rightarrow X$
R2: $X + 0 \rightarrow X$
R3: $-X + X \rightarrow 0$
R4: $(X + Y) + Z \rightarrow X + (Y + Z)$

(a) Subset of group axioms for addition, directed using KBO

$$\begin{tikzcd}
\boxed{(- - a + -a) + a} \arrow[d, "R4"'] \\
\boxed{- - a + ((-a) + a)} \arrow[d, "R3"'] \\
\boxed{- - a + 0} \arrow[d, "R2"'] \\
\boxed{- - a}
\end{tikzcd}$$

(b) Derivation of $- - a$ from $(- - a + -a) + a$

Figure 2.4: Example: derivation of $- - a$ from $(- - a + -a) + a$ using a subset of the group axioms for addition, directed according to KBO.

At this point, no more rules can be applied. While $- - a$ is smaller than $(- - a + -a) + a$, we still need to prove $- - a = a$. To do this, however, the current rule set is insufficient. In a case like this, the KBC algorithm can be used to generate new rewrite rules.

To derive new rewrite rules, KBC uses a concept called *superposition.* For this, the left-hand side of a rewrite rule is matched on the left-hand side of another rule. If a match exists for two rules, a new term can be constructed through unification.

This term can then be rewritten using the two rules it was constructed with. The result of these rewrites is a pair of terms, which must be equal, since they were derived from the same term using existing rewrite rules. Such a pair is called a *critical pair.* If the critical pair is not trivially equal after normalizing, i.e, applying the existing rewrite rules, it is certain that the terms are equal but cannot be proven equal using the existing rules. Therefore, the critical pair is ordered, using KBO, and introduced as a new rule (Dick, 1991).

Figure 2.5 illustrates this process, by deriving the rule $- - a \rightarrow a$ using superposition of rules from figure 2.4a.

The rule derived in figure 2.5b now enables a rewrite from $- - a$ to a. This derivation step completes the proof from figure 2.4. In fact, with the extended rule set, proving $(- - a + -a) + a = a$ becomes trivial, since matching rules can be applied in an arbitrary order to reach the normal form. By running KBC sufficiently long, i.e., continuously generating more rewrite rules, this eventually holds for all pairs of terms that are equivalent with respect to the underlying axioms.

A TRS with this property is called a *confluent* TRS. Formally, confluence denotes that any two terms that can be derived from the same term can be simplified to a common descendant. It directly follows, then, that to find the normal form for a given term, it suffices to apply rewrite rules exhaustively in any order.

Whether a finite confluent TRS exists depends on the given set of axioms. If it does exist, KBC will eventually stop generating critical pairs that, after normalization, are not trivially equal. At this point, the algorithm terminates. If no finite confluent TRS exists, KBC diverges. When trying to prove theorems, divergence turns KBC into a semi-decision procedure (Dick, 1991). That is, any two terms that are in fact equal will eventually be proven to be equal.

Rewrite rules:

R1: $0 + X \to X$
R2: $X + 0 \to X$
R3: $-X + X \to 0$
R4: $(X + Y) + Z \to X + (Y + Z)$
R5: $-X + (X + Z) \to Z$

$$\begin{tikzcd}
& \boxed{(-X+X)+Z} \arrow[dl, "R4"'] \arrow[dr, "R3"] & \\
\boxed{-X+(X+Z)} \arrow[drr, dashed, "R5"] & & \boxed{0+Z} \arrow[d, "R1"] \\
& & \boxed{Z}
\end{tikzcd}$$

(a) Derivation of $-X + (X + Z) \to Z$ by superposition of R3 and R4.

Rewrite rules:

R1: $0 + X \to X$
R2: $X + 0 \to X$
R3: $-X + X \to 0$
R4: $(X + Y) + Z \to X + (Y + Z)$
R5: $-X + (X + Z) \to Z$
R6: $- - X \to X$

$$\begin{tikzcd}
\boxed{- - X+(-X+X)} \arrow[d, "R3"'] \arrow[ddr, "R5"] & \\
\boxed{- - X+0} \arrow[d, "R2"'] & \\
\boxed{- - X} \arrow[r, dashed, "R6"] & \boxed{X}
\end{tikzcd}$$

(b) Derivation of $- - X \to X$ by superposition of R3 and R5.

Figure 2.5: Example: derivation of rules $-X + (X + Z) \to Z$ and $- - X \to X$ using superposition. Matched subterms are marked red, derived rules are marked gray.

3 Methodology

This chapter describes the experimental methodology used to evaluate the effectiveness of *Knuth–Bendix Completion* (KBC) for generating rewrite systems for rewrite-based program optimization techniques. The evaluation is conducted using two frameworks: one based on *equality saturation* (EqSat) using the `egg` library[1] (Willsey et al., 2021b), and one based on a custom *greedy rewriting engine*.

The following sections describe the complete process in detail. Section 3.1 explains how we generated the rewrite rules using KBC, starting from an initial, handwritten rule set. Section 3.2 outlines the procedure for generating random test terms from arithmetic. Section 3.3 introduces the test environment and the implementation of both evaluation frameworks. Finally, section 3.4 summarizes the experimental setup, detailing which combinations of rule sets, rewriting techniques, and test sets we used in the experiments.

3.1 Rule Generation

This section explains the steps taken to generate different rule sets from one initial handwritten rule set using KBC. The first step is the translation of rules from the format used by `egg` to `TPTP`[2] format. The translated rules are then completed using the `twee` theorem prover (Smallbone, 2021). After some postprocessing, the resulting rule set is translated back to the original format.

Section 3.1.1 explains the rationale for choosing `twee` and how we used it to generate rule sets, including its handling of conditional rewrites. Section 3.1.2 discusses postprocessing steps applied to `twee`'s output to make it compatible with `egg`. Finally, section 3.1.3 provides an overview of the different rule sets generated for testing.

3.1.1 Completion Using `twee`

`twee` is an automated theorem prover based on unfailing KBC. It has a number of features that make it convenient to use as a tool for rule generation.

[1]Version 0.10.0
[2]`https://www.tptp.org/`

As input, twee takes a set of axioms in TPTP format. Additionally, it requires a proof goal. Assuming consistency of the input axioms, twee either generates new rules indefinitely or simply runs until termination. Termination may occur when a confluent TRS is found or when specific conditions are met.

Termination Conditions

Termination conditions within twee include a limit on the number of rules generated and on the size of newly generated rules. Therefore, by providing a contradiction as the proof goal and a termination condition, generating different rule sets based on rule set or term size becomes simple. Upon terminating, twee outputs the final rule set, as well as a list of all rules generated in the process.

Ordering and Critical Pair Selection

Another benefit of twee is the ordering it uses. For KBO, it utilizes a weight function, where all function symbols have weight 1 and are ordered based on the frequency with which they appear in the input axioms. While this approach might not be optimal with certain input problems, it is flexible and does not require the user to specify an ordering.

Yet another useful aspect is the way twee selects critical pairs. For this, a scoring function is used. In essence, rules with small left-hand sides have priority. In addition, critical pairs that are closer to the original axioms, i.e., derived in fewer steps, also receive higher priority. The result is that eventually all axioms are used, and rules with smaller left-hand sides are generated first. The latter increases the likelihood of more general rules being generated.

Conditional Rewrite Rules

Possibly the biggest strength twee offers for the purpose of this work is its integration of conditional rewrite rules. Such rules are important when performing optimizations. Consider, for example, the division operation. It is a relatively costly operation to perform and is part of many known identities that can be used for simplification. However, division is only defined if the denominator is unequal to 0. A rule such as $\frac{x}{x} \to 1$ would not only be unsound on its own, it would also enable KBC to generate further unsound rules.

Restricting rewrites during the simplification process is fairly simple, since we can check conditions before rewriting and only simplify if they are provably met. Such semantics-based checks are not possible with KBC. Therefore, conditions must be encoded into the terms directly.

The encoding used by twee was defined by Claessen and Smallbone (2021). They effectively define an if-then-else clause $\texttt{ifeq}(x, y, z, w)$ which reads as if

$x = y$ `then` z `else` w. If such a term exists in the axiom set, the special rule `ifeq`$(x, x, y, z) \to y$ is added. This special rule can be matched on a rule containing an `if-then-else` clause exactly if the condition is fulfilled.

This encoding is what `twee` uses internally. The user can define restricted axioms as implications, i.e. $x = y \implies z = w$, which are then translated. Figure 3.1b shows the encoding at the example of $\frac{x}{x} \to 1$.

`egg` **Encoding**

```
rw!("cancel-div"; "(/ ?x ?x)"
=> "1" if is_not_zero("?x"))
```

(a) Conditional rewrite as expressed in `egg`.

`twee` **Encoding**

`ifeq`(`not_zero`(x), `true`, $x/x, 1) \to 1$,
`ifeq`$(x, x, y, z) \to y$

(b) Conditional rewrite encoded using `ifeq` as defined by Claessen and Smallbone (2021).

Figure 3.1: Encoding of the conditional rewrite rule $\frac{x}{x} \to 1$ in `egg` (left) and its equivalent form in `twee` using the `ifeq` encoding (right).

It is important to note that the rule in figure 3.1b can only become unconditional if another rule `not_zero`$(x) \to$ `true` exists. While this may be given as an assumption in a specific proof setting, it will not be the case when trying to derive general rewrite rules.

However, it is still possible to produce superpositions with this rule, since x/x is simply a subterm of the entire left-hand side. Any rule resulting from such a match would also have the condition attached to it.

3.1.2 Postprocessing of Completed Rules

There are generally two possible outcomes when attempting to complete a set of rewrite rules using KBC (Dick, 1991). The first possibility is that a finite confluent TRS is found. In this case, rule normalization is simple, as any application order will lead to a normalized term. The only thing one has to consider, in particular when using unfailing KBC, is that not all rules are directed in general.

A simple example for this is the equation $x + x = (1 + 1) \cdot x$. KBO cannot order this equation despite the weight of $x + x$ being lower. The reason is that KBO requires the smaller term to have at most as many variables and occurrences of any given variable as the larger term (Baader and Nipkow, 1998).

This is unproblematic in practice since ground terms, i.e., fully instantiated terms, are always orderable under KBO, because x would be instantiated with a term containing only function symbols and constants. Therefore, both sides have the same number of variable occurrences, namely 0. When normalizing a term, one can simply apply KBO on the rewritten term and the original to determine whether the application results in simplification.

Non-confluent Systems and EqSat

If no confluent TRS is found, and termination is forced in another way, we cannot assume confluence. In this case, successful normalization is not guaranteed, even when handling unorderable rules appropriately. One issue that arises due to the lack of confluence is that the rewrite process might "get stuck" at a non-optimal term. This problem can be solved by using a technique such as EqSat, which applies all possible rules at all points in the rewrite sequence.

However, even when ensuring that all possible rewrites are considered, proper normalization is not guaranteed. This is because KBC initially reduces the power of its rewrite system when forcing a direction on initially bidirectional input axioms. If completion succeeds, this initial loss of rewriting power is compensated by the confluence property obtained through additional rules. Again, confluence cannot be assumed if completion does not succeed.

To properly address the issues discussed in this section, this work introduces modified versions of each completed rule set. To address the issue of unorderable rules, for each rule set, we create two different variants. In the first variant, unorderable rules are removed entirely. In the second variant, such rules are replaced by two new rules each, one for each of the possible directions.

To examine the impact of forcing a direction for all rules, we again introduce two different variants of each rule set. The first one simply contains the rules generated by KBC and therefore only simplifying rules. The second variant additionally includes all rules that were contained in the original input set. This variant extends the original rule set rather than replacing it and is therefore guaranteed to be at least as powerful as the original when used with EqSat.

This gives us a total of four versions of each rule set. A comprehensive overview of all rule sets is given in the following section.

3.1.3 Generated Rule Set Variants

The process of generating a rule set using `twee` is straightforward and depends only on the initial set of rules given as input, the termination condition, and the postprocessing detailed in the previous section. However, `twee` is not designed to generate rule sets for program optimization, which is why certain decisions can be made to improve the resulting rule sets. This section introduces some of these decisions and the rule sets that result from them. Finally, it will summarize which sets we used during the tests.

Input Rule Sets

The first input factor we have to consider is the set of input rules. The basis of all sets of input rules was taken from the `math` example provided as part of the `egg` library's test suite (Willsey et al., 2021a). The full rule set can be found in appendix 1.1. The rule set contains, aside from identities used for simplification, some general rules for integration and differentiation. The latter rules were removed for use in this work.

The original rules also contain negation of variables only through multiplication with the constant -1. In combination with `twee`'s term ordering, this may lead to rules such as $x + 1 \to x - (-1)$. This is not an issue when dealing with proofs, but is counterintuitive in the context of program optimization. A simple workaround for this is replacing -1 with $\text{neg}(1)$. This does not change the behavior of the completion algorithm, since there are no rules including a neg operator. It does, however, enforce an ordering which produces $x - \text{neg}(1) \to x + 1$. After completion, this change can simply be reverted to ensure unchanged behavior with `egg`'s EqSat implementation.
Based on this, we created four input rule sets for completion, all of which use the aforementioned workaround for negation. The sets include:

1. the original set,
2. a version without partial functions (division and exponentiation),
3. a version where partial operators were completed separately, and
4. a version where partial operators were completed separately, but all rules derived during KBC were used rather than only the final TRS.

Rule set (2), which we will refer to as `math_no_diff_int_no_div_no_pow` or `no_div_no_pow` for short, does not contain conditional rewrites. We introduce to enable tests that do not depend on `twee`'s special encoding. Since the comparison with rule sets containing division and exponentiation is not very meaningful, there will be sets of test terms that also do not contain these operations as detailed in section 3.2.

Rule set (3), named `math_no_diff_int_sep_div` or `sep_div` for short, should represent a better way of dealing with conditional rewrites. For this set, we selected a number of rules containing division, multiplication, and exponentiation completed them separately. Crucially, rules that contain 0 or subtraction were not considered. This way, no conditions are needed during completion, since no critical pairs that contain a division by 0 can be constructed.

After completing this subset, a simple program added conditions to the rules where needed. The program simply adds a `not_zero` check to all variables appearing inside the denominator of a division, or inside the base of an exponentiation, if the exponent is not known to be a positive constant.

This way, 27 rules were added to the original set. The termination condition for the completion process was that a maximum of 35 rules may be derived. The difference in the numbers results from `twee` removing rules which are already subsumed by others, i.e., where both sides can be normalized to the same term using the remaining rules. We chose a limit of 35 based on the observation that, by this point, a sufficient number of useful rules had been generated.

Rule set (4), which we call `math_no_diff_int_sep_div_plus` or `sep_div_plus` is identical to `sep_div`, except that all 35 generated rules were added. This is based on the observation that during the completion, certain rules might be subsumed by others, but not after introducing conditions.

Rule sets (1) to (4) can be found in appendix 1.

Rule Set Overview

So far we have discussed different input rule sets as well as different ways to process the final rules produced by `twee`. Therefore, the only factor that was not discussed is the termination condition for KBC. The main candidates considered as termination conditions for this work were the total number of rules generated and the size of the left-hand sides of newly generated rules.

We explored both possibilities in preliminary tests, which revealed that terminating based on term size is rather impractical, since the number of derivable rules explodes quickly when increasing the maximum size. Therefore, we generated different rule sets based on the maximum number of rules `twee` should derive. The specific limits used are 40, 60, 80, 100, 150, 200, 1000, and 2500. Recall, however, that during completion, rules may be removed. Hence, rule sets generally contain fewer rules than their specified limit.

The following table 3.1 shows all generated rule sets.

Base Rule Set	Variant	Limits (rules)
math_no_diff_int	(base)	40, 60, 80, 100, 150, 200, 1000, 2500
	no_unorderable	40, 60, 80, 100, 150, 200, 1000, 2500
	extending	40, 60, 80, 100, 150, 200, 1000, 2500
	extending_no_unorderable	40, 60, 80, 100, 150, 200, 1000, 2500
math_no_pow_no_div	(base)	40, 60, 80, 100, 150, 200, 1000, 2500
	no_unorderable	40, 60, 80, 100, 150, 200, 1000, 2500
	extending	40, 60, 80, 100, 150, 200, 1000, 2500
	extending_no_unorderable	40, 60, 80, 100, 150, 200, 1000, 2500
math_sep_div	(base)	40, 60, 80, 100, 150, 200, 1000, 2500
	no_unorderable	40, 60, 80, 100, 150, 200, 1000, 2500
	extending	40, 60, 80, 100, 150, 200, 1000, 2500
	extending_no_unorderable	40, 60, 80, 100, 150, 200, 1000, 2500
math_sep_div_plus	(base)	40, 60, 80, 100, 150, 200, 1000, 2500
	no_unorderable	40, 60, 80, 100, 150, 200, 1000, 2500
	extending	40, 60, 80, 100, 150, 200, 1000, 2500
	extending_no_unorderable	40, 60, 80, 100, 150, 200, 1000, 2500

Table 3.1: Overview of all generated rule sets. Each base rule set produces four variants, including the base itself, and each variant is generated with six completion limits.

3.2 Term Generation

As benchmark data, this work uses randomly generated terms from arithmetic. An important advantage of using this domain is that flaws in the approach, such as inconsistencies in the rule sets, are relatively easy to detect. Further, KBC is known to work well with equational theories from arithmetic, and KBO mostly coincides with the expression simplification goal often used in program optimization.

There are a total of six sets of terms, which we use for both EqSat and greedy rewriting. The following sections describe the different kinds of sets as well as the method used to generate individual terms.

3.2.1 Function Symbols Used in Test Terms

This work uses two different kinds of terms. The first type, which we will refer to as `random_terms` is derived directly from the basic rule set taken from `egg`'s test suite. The function symbols include the constants 0, 1, and -1, variables labeled a to e, and the operators for *addition*, *subtraction*, *multiplication*, *division*, and *exponentiation.*

The second type, referred to as `no_div_no_pow_random_terms` shares the same set of function symbols, except division and exponentiation. The sets using this type of terms, similarly to the `no_div_no_pow` rule sets, serve as a way to test the effectiveness of the generated rule sets, when no conditional rules are required.

3.2.2 Implementation

The program responsible for term generation first creates a specified number of individual terms. These terms are stored in a vector of sets, where each set contains all terms whose length corresponds to the index of the set. The *length* of a term refers to the number of binary operations it contains. Afterwards, random samples of specific sizes are extracted from different sets in the vector to be used as test sets.

The generation of individual terms is handled by a simple recursive function. In the first step, an element from a set containing the function symbols and *"const"* and *"var"* is selected. Depending on the result, either a constant or variable name is chosen at random and returned, or a function symbol is selected at random, resulting in two recursive calls to determine its arguments.

When selecting variable names, there is a bias towards existing variables, as well as a limit on the number of distinct variables a term can contain. This increases the likelihood that the resulting terms are simplifiable, thereby producing more meaningful evaluation data.

3.2.3 Test Set Overview

Table 3.2 lists the different test sets we generated, including the number and size of terms they contain. During the tests, the different term sizes highlight if and how the performance of different approaches and rule sets depends on the size of the input term.

Term Set	Variant	Number of Terms	Term Size Range
`random_terms`	`small`	1000	1–3
	`large`	500	10
	`huge`	250	25
`no_div_no_pow_random_terms`	`small`	1000	1–3
	`large`	500	10
	`huge`	250	25

Table 3.2: Overview of all generated term sets. Each term set is produced in three size variants (`small`, `large`, and `huge`) differing in the number of terms and their size.

3.3 Test Environment

This work evaluates the effectiveness of KBC-generated rewrite rules in two ways. The first series of tests uses EqSat to simplify input terms using the `egg` library, while the second series applies a custom greedy rewriting engine for the same purpose.

Effectiveness is measured as the average reduction in symbol count between input and output terms. For instance, if the input term is $a - a + b$ and the simplified output is b, the difference amounts to $5 - 1 = 4$ symbols. Thus, for each technique and rule set combination, the mean difference across all input terms indicates its overall simplification performance.

In the greedy approach, only the final simplified output is recorded. Under EqSat, however, each rule set is evaluated across multiple time limits, as unrestricted E-graph growth can quickly exceed available memory. Varying the time limits also provides additional insight into rule-set efficiency: if, for example, two rule sets, *R1* and *R2*, reach the same optimal term, but *R1* does so more quickly, then *R1* would be considered more practical.

3.3.1 Equality Saturation

This work uses the `egg` library for equality saturation. In addition to input rules and expressions to simplify, `egg`'s E-graph runner requires a language (i.e., a set of operators and symbols) and an analysis, which defines additional transformations within the E-graph beyond rewrite rules.

The experiments use the language and analysis from `egg`'s `math` example. The included analysis performs *constant folding*, enabling the E-graph to directly evaluate expressions consisting only of constants.

For rule scheduling `egg`'s `SimpleScheduler`, which imposes no restrictions on rule application. To reduce the likelihood of exceeding the available memory, we capped the maximum number of E-nodes at 1,000,000. We ran EqSat using the following time limits: 0.0001s, 0.0005s, 0.001s, 0.005s, 0.01s, 0.05s, 0.1s, 0.5s, and 1s. Because the runner only checks these limits after completing an iteration, actual simplification times may exceed the nominal limits significantly if an iteration is particularly long.

3.3.2 Greedy Rewriting

In contrast to EqSat, greedy rewriting does not explore all possible rewrite paths. Instead, it applies rules sequentially, according to some strategy, until no more rules can be matched. This makes the approach considerably less resource-intensive compared to EqSat. The downside, however, is that the effectiveness of greedy rewriting depends much more on the selection of rewrite rules.

The approach resembles the idea of a KBC-based proof as discussed in section 2.3, where the input term should be proven equivalent to its normal form. The difference to that proof scenario is mainly that during a KBC proof, arbitrarily many rules may be generated, while an optimization procedure might require the rules to be known beforehand. Even if we were to generate new rules during completion, it would not be possible to assert that the simplification process is complete, as the normal form is obviously unknown in practice.

Therefore, EqSat and greedy rewriting represent a trade-off in terms of resource consumption and coverage of the search space of equivalent terms. If the input rule set is a confluent TRS, we can expect the trade-off to disappear, since both approaches would find the optimal term, but the greedy approach would require significantly less time and memory. However, since a finite confluent TRS most likely does not exist in most cases, and specifically in the domain used in this work, we must assume that there exist at least some terms for which greedy rewriting does not find the normal form.

Consequently, this test series aims to assess whether KBC-generated rewrite rules can make greedy rewriting a viable alternative to EqSat in scenarios where minimizing running time and memory usage is essential.

Implementation

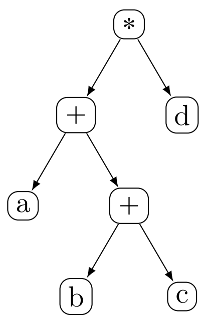
To assert that the test results properly reflect the impact of rule sets and are neither positively nor negatively impacted by the specifics of existing rewriting engines, we use a custom rewrite engine for this work. The implementation is specialized for the domain and KBC-generated rule sets, but is not optimized for running time. Given the difference in complexity when comparing EqSat and greedy rewriting, such optimizations would add little value, and performing them is a demanding task on its own.

Parsing and Term Representation. The entire process of simplifying a set of input terms using a set of rewrite rules is handled by a single `rust` program. The required format for the rule set is the same as that for EqSat using `egg`, although the rules are parsed as strings.

During parsing, each input rule is converted into an internal `struct` consisting of three fields: the left-hand side, the right-hand side, and an optional condition.

Terms are represented as flatterms, or arrays of symbols, where a symbol corresponds to a tuple consisting of an identifier string and a number, representing the length of the subterm rooted in the symbol itself. This representation is also used in `twee` and enables efficient matching and rewriting as term traversal happens on an array rather than a tree structure. Figure 3.2 illustrates the term representation using an example.

Tree representation



Flatterm representation

$(*, 7)$	$(+, 5)$	$(a, 1)$	$(+, 3)$	$(b, 1)$	$(c, 1)$	$(d, 1)$

Figure 3.2: Tree and flatterm representation of the term $(a + (b + c)) * d$. Each symbol, i.e., tuple `(id, size)` stores a string and the size of the subterm rooted at that symbol.

After being translated, rules are handled in one of three ways, depending on a comparison of the sizes of their left and right-hand sides:

- $|\text{lhs}| > |\text{rhs}|$: Rule is added to the set of simplifying rewrite rules.
- $|\text{lhs}| = |\text{rhs}|$: Rule is added to the set of *canonicalizers*.
- $|\text{lhs}| < |\text{rhs}|$: Rule is deleted as it contradicts KBO. This only happens if the input rule set is extending (see section 3.1.2).

Eliminating rules that contradict KBO is important, as termination depends on only applying simplifying rules. For the same reason, rules that are not strictly simplifying require special treatment. These rules are added to the set of canonicalizers and correspond to rules which are only orderable under KBO when instantiated on ground terms, as discussed in section 2.3. Their role in the simplification process is explained later in this section.

Rewriting Procedure. After parsing the inputs, terms are being rewritten sequentially. Algorithm 1 gives a high-level overview of the rewriting process for a single term. The following paragraphs provide detailed descriptions of the functions RewriteOneStep, FoldConstants, and Canonicalize, in which transformations are applied.

Algorithm 1 Greedy Term Rewriting Procedure

```
Require: Rule list $rules$, canonicalizer list $canon$, term $term$
Ensure: Rewritten term $t'$
 1: $currentTerm \leftarrow term$
 2: $canonicalizable \leftarrow$ true
 3: while $canonicalizable$ do
 4:     while RewriteOneStep($rules, currentTerm$) returns $Some(newTerm)$ do
 5:         $currentTerm \leftarrow newTerm$
 6:         FoldConstants($currentTerm$)
 7:     end while
 8:     $canonicalizable \leftarrow$ Canonicalize($currentTerm, canon$)
 9:     FoldConstants($currentTerm$)
10: end while
11: return $currentTerm$
```

Individual Rewrite Steps. Transforming the current term by applying a single simplifying rewrite is handled by the function RewriteOneStep. This function simply iterates the set of strictly simplifying rewrite rules until some rule can be matched. If none of the conditions attached to the rule are violated, the current term is rewritten accordingly, and the result is returned.

Whether a rule can be matched on a given term t is determined by a unification function. This function first identifies the indices of t that contain the same symbol as the root of the left-hand side *lhs* of the rewrite rule. If such indices exist, the function iterates over them left to right and attempts to match subterms. To keep track of variable assignments, it maintains a substitution map *subst*, mapping variable names to terms.

Matching subterms works by iterating over *lhs* and at each step handling one of the following cases depending on the current symbol:

- Function symbol: Assert the same symbol on *t*. Advance one symbol in *lhs* and *t*.
- Variable *var*: Read the subterm *s* rooted at the current symbol of the *t*. If *subst* contains *var* assert that *s* matches the current value. If *var* is not yet present in *subst*, add it with *s* as its value. Advance one symbol in *lhs* and the length of *s* on *t*.
- Constant *c*: Assert that *c* equals the current symbol of *t*. Advance *lhs* and *t* by one symbol.

If for some root index *idx* all assertions pass, `Some`((*subst, idx*)) is returned; otherwise `None` is returned.

If unification succeeds, the variables in the right-hand side of the rule whose left-hand side was matched are replaced according to the substitution mapping returned by the unification function. The subterm rooted at *idx* is then replaced by the right-hand side, and affected subterm sizes are updated, completing the rewrite step.

Constant Folding. Constant folding is technically not required in this context, since rewrites happened based on the structure of the term. However, since `egg` uses constant folding during EqSat, it is required to enable a fair comparison between the methods.

The implementation of constant folding is fairly straightforward. The FOLDCONSTANTS function iterates over the current term and, for each addition, subtraction, multiplication, and division, checks whether both operands are constants. In the case of division, the function additionally asserts that the denominator is not trivially equal to 0. If these conditions are fulfilled, the root symbol is replaced with the result, and relevant subterm sizes are updated.

The logic behind this function is identical to the constant folding analysis used by `egg`.

Canonicalization. When applying rules sequentially, certain rules, such as commutativity, are challenging to deal with. When applying commutativity to transform a term *t* to a term *t'*, for example, it is guaranteed that the same rule can be applied again to transform *t'* back to *t*. This may lead to indefinitely repeating chains of rewrites. However, rules of this kind often play an important role in enabling rewrites. If, for instance, we want to simplify the term $0 + x$ and we only have the rule $x + 0 \rightarrow x$, commutativity is necessary to find the normal form.

KBC solves this by introducing canonical representations through lexicographical ordering. Since this principle is applied during rule generation, it also has to be applied during the rewrite process. Therefore, the Canonicalize function implements the canonicalization of terms based on KBO.

The canonicalization process iterates over all rules and attempts to match them on a given term. Similarly to the RewriteOneStep function, it utilizes a unification function. However, rather than identifying a single match by searching from left to right, i.e., top-down in the expression tree, during canonicalization, matches are searched right to left, or bottom-up, and all matches are recorded. This way, the entire term is canonicalized, with respect to a given rule, in a single iteration.

If the unification function finds a match and rewrite conditions are not violated, the substitution identified by the unification function is applied. Before applying the final rewrite, however, a comparison function checks whether the rewritten term is actually smaller according to an ordering implementing KBO. If for any match of any rule this condition is fulfilled and a rewrite is performed, Canonicalize returns `true`, otherwise it returns `false` to indicate that the term is already fully canonicalized.

3.4 Test Setup

This section summarizes the experimental setup used for evaluating the rule sets. It describes the combinations of rule and test sets applied in both EqSat and greedy rewriting, as well as the data recorded during the experiments.

3.4.1 Combinations Tested

We evaluated both rewriting methods on all six test sets discussed in section 3.2 and all rule set variants introduced in section 3.1.

For EqSat, we included all rule sets with sizes 40, 100, 150, and 200. For the greedy approach, we tested all sizes (i.e., 40, 60, 80, 100, 150, 200, 1000, and 2500). We excluded set sizes 60 and 80 for EqSat because they did not add much insight.

Sizes 1000 and 2500 were used exclusively for the greedy approach, because individual iterations under EqSat would take excessively long with such large rule numbers. This would cause the actual time spent simplifying to significantly exceed the set time limits, invalidating the tests.

3.4.2 Evaluation

In the case of EqSat, we ran all combinations of rule and test sets once per time limit specified in section 3.3.1. During each run, we recorded information about the run, as provided by the `egg` runner itself. This information includes simplification time (split into rule application time, search time, and rebuild time), the number of iterations, and the stop reason (typically the time limit), in addition to the original and simplified term.

By simply counting the symbols inside a term, we determined the cost of each input and output term. The difference between the two then yields the cost saved by simplifying. Adding up these differences for each test set and dividing by the size of the test set gives us the average simplification effectiveness of a given rule set under EqSat on terms of a given complexity under the given time constraint.

For the greedy rewriting approach, each rule and test set combination was executed once. We recorded the total time elapsed while rewriting. We then divided the total time by the number of terms in the test set. In addition, for each term, we recorded the input and output from which we calculated the effectiveness using the same cost function as for EqSat.

4 Results

This chapter presents the experimental results obtained by following the methodology described in chapter 3. The experiments evaluate the effectiveness of Knuth–Bendix Completion (KBC)-generated rewrite rules under different rewriting frameworks.

The data shown here represents a subset of all results, highlighting the most relevant findings for analysis. The full tables underlying the figures presented in this chapter are included in appendix 2.

Section 4.1 first compares the different rule generation methods given in section 3.1 and then compares promising rule sets against `egg`'s handwritten rule set. Section 4.2 shows the impact of different rule set sizes when using greedy rewriting for simplification. Lastly, section 4.3 compares the performance of EqSat and greedy rewriting on the given test sets.

4.1 Equality Saturation

This section presents the results obtained using the equality saturation (EqSat) framework described in section 3.3.1. The goal of these experiments is to evaluate how different KBC-generated rule sets affect the simplification performance of EqSat across different term sizes.

We first analyze how the postprocessing choices introduced in section 3.1.2 affect the EqSat process. This comparison shows the influence of handling unorderable and extending rules on simplification outcomes. We then compare the most promising KBC-generated rule sets to the original handwritten rule set provided by `egg` to assess whether KBC-based rule generation leads to measurable improvements in output quality across different time limits.

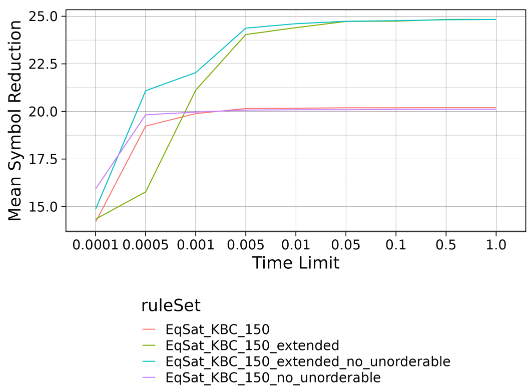
4.1.1 Impact of Postprocessing on Rule Sets

Figure 4.1 shows the simplification effectiveness of rule sets depending on postprocessing as discussed in section 3.1.2. In both subfigures, EqSat was executed on test sets containing terms with 25 binary operators.

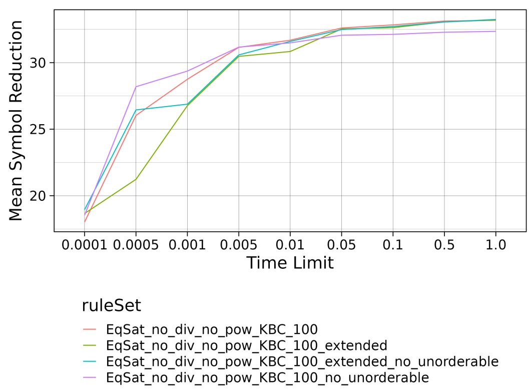
Figure 4.1a shows the case where division and exponentiation were included. We can see that the extending rule sets, i.e., those that are supersets of the original handwritten set, yield significantly better terms than those that only contain the rules generated by KBC after running EqSat for one second.

The plot also reveals that there is virtually no difference between the final outputs of sets that contain unorderable rules generated by KBC and those that had such rules removed. The impact of including unorderable rules becomes apparent, however, when considering shorter time limits, as both rule sets without them outperform their counterparts until the simplification effectiveness plateaus.

In figure 4.1b, where division and exponentiation were excluded in both rule and test sets, the impact of unorderable rules on shorter time limits shows in a similar way. In this instance, however, including unorderable rules improves performance on non-extending sets, such that only the non-extending rule set without unorderable rules performs significantly worse at longer time limits.



(a) Results including division and exponentiation.



(b) Results excluding division and exponentiation.

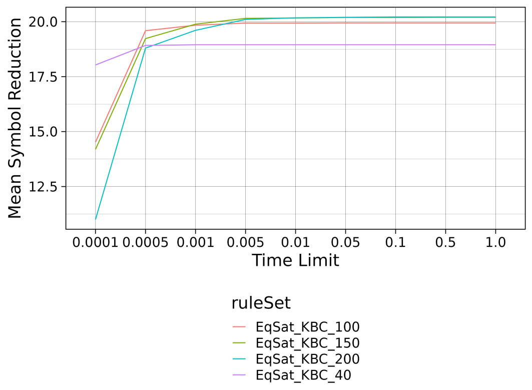
Figure 4.1: Comparison of EqSat results by rule set postprocessing on terms containing 25 binary operators. Subfigure (a) shows results for rule and test sets containing division and exponentiation, while subfigure (b) excludes these operations. Higher values indicate more effective simplification.

4.1.2 Impact of Rule Set Size

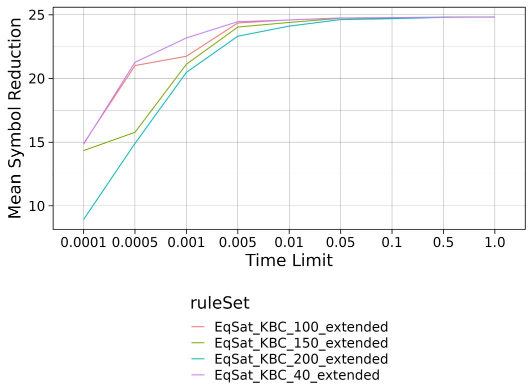
Figure 4.2 shows the simplification effectiveness of EqSat using KBC-generated rule sets as a function of rule set size. In both subfigures, EqSat was executed on test sets containing terms with 25 binary operators.

Figure 4.2a shows the performance of rule sets that contain only rules that `twee` generated. On one hand, we see a clear positive correlation between rule set size and simplification effectiveness after one second. On the other hand, we see a negative correlation between rule set size and simplification performance at shorter time limits.

The negative impact of increasing the number of rules on the performance at shorter time limits is also visible in figure 4.2b, which uses extended versions of the same rule sets. In contrast to the case of non-extending sets, however, an increase in the number of rules does not lead to significantly higher simplification effectiveness.



(a) Non-extended rule sets.



(b) Extended rule sets.

Figure 4.2: EqSat simplification effectiveness across different rule set sizes. Subfigure (a) shows results for non-extended rule sets, while subfigure (b) shows results for their extended counterparts. Higher values indicate more effective simplification.

4.1.3 Comparison with the Handwritten Rule Set

After assessing the impact of postprocessing choices and rule set size, we now compare the KBC-generated rule sets with the best performance against `egg`'s original handwritten rule set.

Figures 4.3 – 4.5 show the performance of selected rule sets on the test sets introduced in section 3.2.

The subfigures focusing on terms including division and exponentiation include the results of the handwritten rule set as well as completed versions based on the input rule sets introduced in section 3.1.3, which contain all operators. The sets were completed using `twee` with a limit of 150 generated rules. In accordance with the results from the tests in the last two sections, the rule sets are extended and do not include unorderable rules.

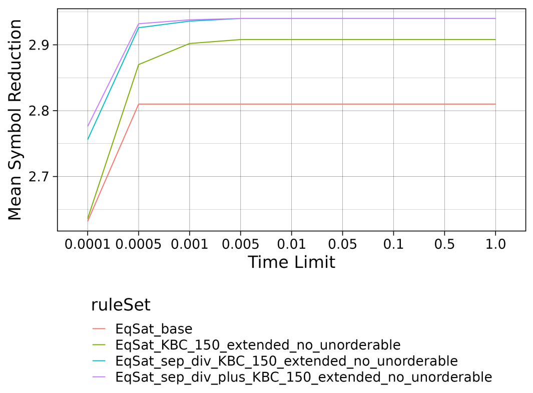
The subfigures showing the results without division and exponentiation use `egg`'s basic rule set with the operations removed and different completed versions of the same set. The sets were completed using `twee` with a limit of 100 generated rules and include all postprocessing variations.

Figure 4.3 shows the performance of different rule sets on small input terms with one to three binary operators.

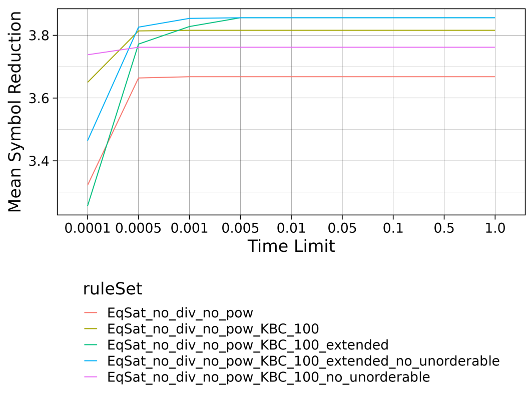
Both subfigures show that all KBC-generated rule sets outperform the handwritten rule set at almost all time limits. The KBC-generated sets also take longer to find their respective optimal terms.

Subfigure 4.3b also confirms that the observations from section 4.1.1 hold independent of the size of input terms.

Subfigure 4.3a gives additional insight into the effect of the input rule set choices for KBC. We can see that completing rules that are relevant for division and exponentiation separately leads to a substantial improvement in simplification effectiveness across all time limits.



(a) With division and exponentiation.

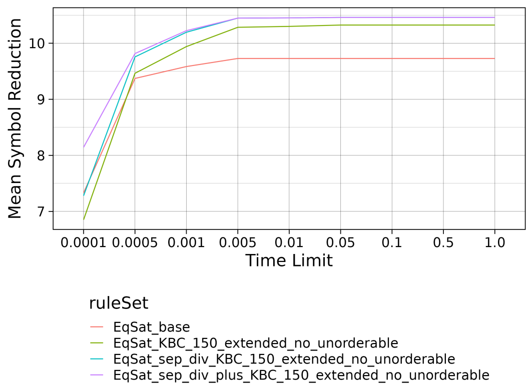


(b) Without division and exponentiation.

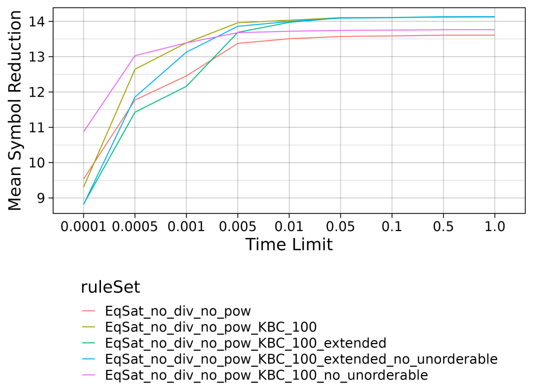
Figure 4.3: EqSat simplification effectiveness for small terms (1–3 binary operators). Subfigure (a) shows results for rule and test sets containing division and exponentiation, while subfigure (b) excludes these operations. Higher values indicate more effective simplification.

Figure 4.4, for the most part, confirms the observations made on small terms. A notable difference seen in subfigure 4.4a is that the handwritten rule set outperforms two of the KBC-generated rule sets at the time limit of 0.0001s.

Figure 4.5 again shows similar results. An observation we can make in both subfigures is that almost all KBC-generated rule sets show less of an improvement when going from time limit 0.0005s to 0.001s. When division and exponentiation are excluded, the handwritten rules show this as well. Note also that the line for *sep_ div_ plus* rule set ends at time limit 0.5s. This is due to EqSat exceeding the allocated memory limit of 16 GB for some term in the test set.

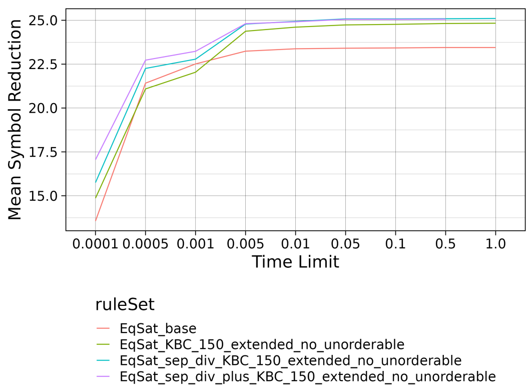


(a) With division and exponentiation.

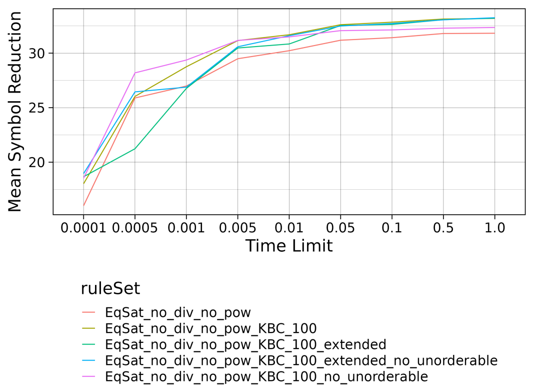


(b) Without division and exponentiation.

Figure 4.4: EqSat simplification effectiveness for large terms (10 binary operators). Subfigure (a) shows results for rule and test sets containing division and exponentiation, while subfigure (b) excludes these operations.



(a) With division and exponentiation.



(b) Without division and exponentiation.

Figure 4.5: EqSat simplification effectiveness for huge terms (25 binary operators). Subfigure (a) shows results for rule and test sets containing division and exponentiation, while subfigure (b) excludes these operations.

4.1.4 Summary

In summary, certain KBC-generated rule sets outperform `egg`'s handwritten rule set but generally converge to a consistent solution more slowly. This holds consistently across all sizes of input terms, independent of whether division and exponentiation are included in the rule and test sets.

We also saw that purely KBC-generated rule sets have inferior performance compared to those that keep the original handwritten rules. For the latter, increasing the number of rewrite rules slows EqSat's convergence without providing a meaningful improvement in simplification effectiveness.

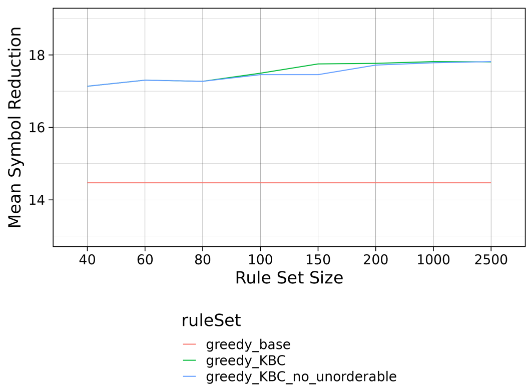
4.2 Greedy Rewriting

This section presents the results on simplification effectiveness when using greedy rewriting. The approach was tested as described in section 3.4. However, extended and non-extended rule sets behave almost identically, because preprocessing during the parsing step described in section 3.3.2 removes all rules that would increase term complexity, effectively aligning both variants.

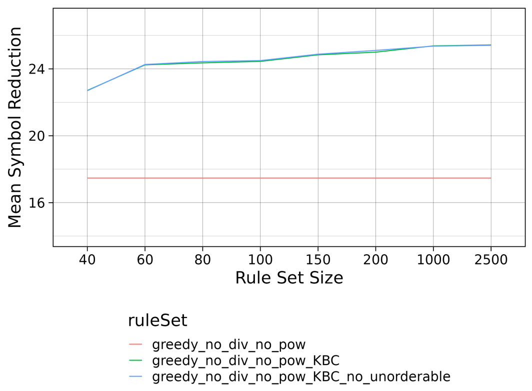
Differences in performance based on the input rule set will be covered in section 4.3. Consequently, this section only covers the impact of rule set size on simplification effectiveness. Figure 4.6 shows this impact for terms of size 25 with and without division and exponentiation.

In both subfigures, we see that both KBC variants perform better than the handwritten rule set under greedy rewriting, regardless of the rule set size. Similarly, we see in both cases that increasing the number of rewrite rules consistently improves rewriting effectiveness, although the increase from 1000 to 2500 rules has little effect.

While including unorderable rules seems to have no impact in general, subfigure 4.6a shows some divergence at rule set size 150.



(a) With division and exponentiation. *greedy_base* represents the original handwritten rule set.



(b) Without division and exponentiation. *greedy_no_div_no_pow* represents the original handwritten rule set.

Figure 4.6: Greedy rewriting simplification effectiveness for huge terms (25 binary operators). Subfigure (a) includes these operations, while subfigure (b) excludes them. Higher values indicate more effective simplification.

4.3 Comparison

Finally, this section uses the experimental results to compare EqSat and greedy rewriting when using KBC-generated rule sets. The results presented here will be the basis for assessing the trade-off between simplification effectiveness and resource consumption, giving insight into the viability of greedy rewriting as an alternative to EqSat.

For each test set, we consider the highest-performing KBC-generated rule set for EqSat and greedy rewriting, as well as the original handwritten rule set provided by `egg`. Table 4.1 summarizes the results. The reported simplification time for EqSat corresponds to the first time limit at which the respective configuration achieved its best simplification result. The validity of the simplification times for both methods will be discussed in detail in section 5.2.1.

Table 4.1: Comparison of best-performing rule sets for EqSat and greedy rewriting versus handwritten rules. EqSat simplification time is the time limit of the first occurrence of the optimal simplification.

Test set	Config.	Avg. Symbol Red.	Simplification Time [s]
no div/pow, huge	G-KBC1000	25.36	0.0003
	E-KBC100	**33.25**	1.0000
	E-base	31.82	1.0000
no div/pow, large	G-KBC1000	10.88	0.00009
	E-KBC100	**14.13**	1.0000
	E-base	13.61	0.5
no div/pow, small	G-KBC1000	3.24	0.000015
	E-KBC100	**3.86**	0.005
	E-base	3.67	0.0005
with div/pow, huge	G-KBC1000	21.50	0.00027
	E-KBC150	**25.10**	1.0000
	E-base	23.45	0.5
with div/pow, large	G-KBC1000	8.92	0.000081
	E-KBC150	**10.46**	0.05
	E-base	9.73	0.005
with div/pow, small	G-KBC1000	2.68	0.000017
	E-KBC150	**2.94**	0.005
	E-base	2.81	0.0005

Legend: G=greedy, E=EqSat; "KBC" indicates KBC-generated rule sets (number = rules). "base" refers to the handwritten `egg` rule set. For EqSat and greedy, KBC sets exclude unorderable rules; EqSat rule sets are extended.

Overall, the table confirms that EqSat consistently achieves higher simplification effectiveness than greedy rewriting. The difference is substantial at roughly 10%-20% when comparing the greedy approach with EqSat using `egg`'s original rules. When using a more optimal rule set for EqSat, the difference becomes even larger.

In terms of running time, the difference is even larger, with greedy rewriting requiring several orders of magnitude less time to reach its optimal solution. While section 4.1 shows that EqSat yields diminishing returns as the time limit increases, section 4.2 shows similar results for increasing the number of rules under greedy rewriting.

The comparison shows that both methods benefit from KBC-generated rule sets, but the trade-off between simplification effectiveness and resource consumption remains.

5 Discussion

This chapter builds on the previous chapters by providing an interpretation of the results. It also critically assesses the methodology we used, under consideration of the theoretical background established in chapter 2. Finally, it provides answers to the research questions this paper set out to answer.

These questions are:

1. Do KBC-generated rule sets improve equality saturation in terms of simplification effectiveness and running time, compared to handwritten rules?

2. Do KBC-generated rule sets enable greedy rewriting as a viable alternative to equality saturation in scenarios where minimizing running time and memory usage is essential?

5.1 Equality Saturation

In this section, we discuss the results presented in section 4.1. This includes the effect of different rule set variants and the resulting implications. We will also address the choice of `egg`'s example rules as base rules and benchmark.

5.1.1 Test Results

Section 4.1.1 showed the difference in simplification effectiveness between rule sets that include the original rules and those that exclude them. This result has rather strong implications for KBC-generated rule sets in general.

During the completion process, KBC reorders the input rules if necessary. In particular, rules that are strictly increasing term complexity are reordered. An example of such a rule, which can be found in the original rule set (see appendix 1.1), is $a - b \rightarrow a + (-1 * b)$. This rule is very important, because it canonicalizes subtraction into addition, enabling addition-specific rules such as commutativity and associativity.

Take, for instance, the term $x + (y - x)$. The only rule we can apply is commutativity on the addition, yielding $(x - y) + x$. If we have the rule $a - b \rightarrow a + (-1 * b)$ at our disposal, EqSat can derive the following rewrite: $x + (y - x) \rightarrow x + (y + (-1 * x)) \rightarrow x + ((-1 * x) + y) \rightarrow (x + (-1 * x)) + y \rightarrow (x - x) + y \rightarrow y$. Of course, under

EqSat these rewrites don't happen sequentially, but since $a - b \to a + (-1 * b)$ exists, $y - x$ and $y + (-1 * x)$ inhabit the same e-class. This enables rewrites on both terms.

If the rule is removed, this equivalence cannot be established, and so the term cannot be simplified. In theory, however, KBC should be able to derive a new rule that manages to simplify $x+(y-x)$ directly. The issue here is that it is not possible to determine beforehand when, during completion, such a rule is generated. While this specific example may be attributed to the lack of a general negation operator in `egg`'s rules, there most likely are many such cases across different domains.

That increasing the number of rewrite rules can mitigate this problem becomes clear from section 4.1.2. However, the effect of adding new rules already becomes minimal upon reaching 150 rules. We again cannot know whether, at some point during KBC, "breakthrough rules" would be generated that significantly increase the rule set's power.

This limitation makes purely KBC-generated rule sets less reliable.

Nevertheless, the results also show that rule sets that contain both the original and KBC-generated rules outperform just the original rules consistently and across different time limits. There might be two reasons why this is the case.

First, the original rule set contains some rules that increase term complexity, but not all rules are bidirectional. Keeping rules unidirectional where possible helps guide EqSat in order to keep the process technically feasible, but it also reduces the overall rewriting power. KBC can compensate for this loss in power by generating new rules. Since its input rules are technically equalities rather than directed rules, KBC can derive rules that allow those rewrites which EqSat misses due to directionality in the rules.

Second, KBC-generated rules are more strongly directed. Where the original rule set might have to increase complexity for some iterations before simplifying, KBC may generate rules that can simplify immediately. KBC may also generate rules that reduce complexity more substantially per rule application.

While the results indicate that KBC-generated rules in isolation are not ideal for EqSat, there are sufficient theoretical arguments to suggest that KBC can be very useful to generate additional rewrite rules, augmenting handwritten ones. The results generated in this work also support this claim.

5.1.2 Choice of Base Rules and Benchmark

Upon examination of simplification results on individual terms produced by EqSat using `egg`'s example rules, it is very much possible to find weaknesses in the rule set. No simplification was achieved, for example, on the input term $a \cdot \frac{1}{\frac{a}{b}}$ which is unsurprising considering that the rule set is missing rules such as $b \neq 0, b \neq 0 \implies$

$\frac{\frac{a}{b}}{c} \to \frac{a \cdot c}{b}$ and $a \neq 0 \implies a \cdot \frac{b}{a} \to b$. Both rules are well-known, and it would be expected that they would either be included directly or be derivable if a rule set is intended for practical use.

twee manages to derive these rules very quickly when completing division-relevant rules separately, as was done for the *sep_ div* and *sep_ div_plus* rule sets, although it does not derive them when just given the original set. This indicates that the encoding for conditional rules in twee appears to hinder rule generation.

Since the egg example rules fail to simplify such basic terms, we should question whether it is appropriate to use them as the benchmark.

While one could argue that it would be possible to manually design a rule set that equals the performance of the KBC-generated rule sets, it is important to note that KBC-generated sets, as long as they are extending, are at least as powerful as the original rules.

More importantly, however, using KBC does not require any domain knowledge. This makes it more broadly applicable. While gaps in arithmetic rule sets are easy to spot, they may be less apparent in domains with more complex axioms and lesser-known identities.

In summary, using arithmetic terms and an unoptimized base rule set does not invalidate the results. However, additional tests are required to confirm that rewrite rule generation for optimization using KBC translates well into other domains.

5.2 Comparison of Greedy Rewriting and Equality Saturation

Section 4.3 reflects the overall findings of this work well and confirms the trade-off between simplification effectiveness and resource consumption we discussed in multiple sections. Whether the results can be considered generalizable is not clear.

Methodological concerns can be raised with the reported simplification times, as well as the way rule generation with KBC was implemented with egg's example rules.

5.2.1 Simplification Time

The difference in simplification time we observed in section 4.3 is substantial. This is unsurprising, given the computational cost associated with matching rules on an e-graph (Zhang et al., 2022), rebuilding it after applying rewrites (Willsey et al., 2021b), and searching and extracting a candidate for the optimal term after saturation terminates (Yin et al., 2025).

There are, however, some factors we need to consider when interpreting the results from this work. The first consideration is that we chose the time limit at which EqSat's simplification effectiveness reached a plateau as its simplification time. This obscures the fact that much earlier during simplification, terms may have been found that are only slightly less optimal.

In particular, when comparing against the output terms, the greedy approach produces, an argument can be made that defining simplification time in this way introduces a bias against EqSat.

Conversely, we should note that the time limits, with which we ran EqSat, do not include extraction time. To add to this, recall from section 3.3.1 that `egg` only checks time limits after iterations. Examining the outputs for individual terms reveals that in some instances, even on small terms, using the original rule set, the total simplification time ends up being multiple times larger than the time limit. We can find a small selection of such examples in appendix 3.

In section 3.3.2, we discussed the implementation of the custom rewriting engine we used to test the greedy approach. We already mentioned in this context that the rewriting engine is poorly optimized. This becomes particularly apparent when we consider the canonicalization function, which the program calls whenever no simplifying rule can be applied.

To canonicalize, the program tries to match every canonicalizer on every subterm. This is very inefficient and can be solved by keeping track of which parts of the term were modified since the last canonicalization run. Especially on large input terms and with rule sets containing many canonicalizers, this hurts running time performance significantly.

We can therefore conclude that, while for both methods the simplification time metric is not fully accurate, the performance benefits of the greedy approach in terms of running time can be assumed.

5.2.2 KBC Implementation and Base Rules

In section 5.1.2, we already focused on the role of rules that increase term complexity under EqSat. In this context, we also briefly mentioned the absence of a general negation operator in the original rule set and how this is less of a problem for EqSat as long as certain rules can bridge between different representations of subtraction.

Section 4.2 explains why this is not possible for the greedy approach, which is why we would need to address this issue differently. One option for doing so might be to simply extend the base rules manually to include general negation. This way, KBC might derive more general rules such as $a - b \to a + (-b)$.

This is, however, rather speculative and would require further testing.

The last observation we discuss here is that completing division and exponentiation separately leads to significantly better simplification effectiveness, as shown in section 4.1.3. This raises the question of whether varying the number of rules completed separately or the subset used could further improve the results.

Alternatively, it would be interesting to examine whether a specialized implementation of KBC for program optimization would be beneficial. Such an implementation would likely include a different way of handling conditional rewrites and the use of specialized weight functions for KBO.

However, further work is needed to assess the viability of such a specialized implementation for program optimization.

5.3 Summary

With respect to the research questions we have formulated, the results produced in this work suggest two key findings:

- Augmenting handwritten rule sets with the help of KBC can enhance the simplification effectiveness of EqSat significantly. Whether the results translate to other domains and whether KBC-generated rules can be effective in isolation is inconclusive.

- Greedy rewriting does not reach the same level of simplification effectiveness as EqSat when using KBC-generated rule sets. The advantage in resource consumption makes it a useful approach regardless. Whether the gap in simplification effectiveness can be closed through adjustments to the initial rule sets remains to be tested.

6 Related Work

To accurately position this work within current research on rewrite-based program optimization, this chapter discusses notable work in the fields of equality saturation (EqSat) and automated rewrite rule generation, with respect to the contributions of this work. Additionally we discuss a specialized variation of EqSat which might fit KBC-generated rules in particular.

6.1 Equality Saturation

EqSat as a tool for program optimization was first introduced by Tate et al. (2011). The authors propose it as a solution to the *phase ordering problem*, which describes the dependence of optimization success on the order in which optimization steps are applied.

The approach has gained popularity in recent years. The `egg` framework (Willsey et al., 2021b) builds on EqSat, addressing the lack of extensibility and high computational cost. It improves extensibility by introducing domain-specific analyses on equivalence classes, and performance by introducing *rebuilding*.

The latter relaxes e-graph invariants such as congruence. More specifically, *rebuilding* allows multiple rewriting steps before reestablishing congruence, i.e., that all equivalent expressions in the e-graph inhabit the same e-class.

In addition to being the basis for EqSat implementations such as `egglog` (Zhang et al., 2023), which combines EqSat with database techniques, `egg` has been integrated in a number of practical applications.

Examples from the domain of program optimization include `Herbie` (Panchekha et al., 2015) and `Felix` (Zhao et al., 2024). `Herbie` is a tool that uses EqSat to reduce rounding errors in floating-point arithmetic. `Felix` optimizes tensor-based programs by approximating performance with differentiable functions, which it simplifies using EqSat.

Notably, applications of EqSat tend to use manually constructed sets of rewrite rules. This seems reasonable because it offers flexibility to balance the trade-off between coverage of the search space of equivalent terms and resource consumption.

A recent survey on synthesis of rewrite rules by Hong (2024) addresses this, summarizing the different approaches that are currently available for rule generation in the context of rewrite-based optimization techniques. The survey includes

KBC (Dick, 1991) as a predecessor of other techniques, but remarks that it is difficult to evaluate its effectiveness in the domain of program optimization.

Providing first insights into the effectiveness of KBC as a tool for generating rewrite rules to be used in program optimization is the aim of this work.

6.2 Rewrite Rule Generation

The individual subprocesses of EqSat have seen significant improvements in recent years. Notable examples include the *relational e-matching* approach by Zhang et al. (2022), which uses *relational join*, a technique used in databases, to give guarantees for the running time of pattern matching on e-graphs, as well as `SmoothE` (Cai et al., 2025), which aims to make term extraction based on complex cost functions feasible.

In the area of automated rewrite rule generation, `Ruler` (Nandi et al., 2021) explores the synthesis of rewrite rules through equality saturation itself. To achieve this, it enumerates terms up to a specific size within an e-graph. It then selects candidates for rewrite rules by evaluating terms on input vectors. If two terms behave identically on all inputs, they are equivalent and their e-classes can be merged. This process happens iteratively with increasing term sizes, until ideally all terms with identical behavior can be proven to be equivalent using the established rule set.

While both `Ruler` and this work, through KBC, generate rule sets for specific domains, there are some important differences.

One of those differences is that `Ruler` does not require any input rules. It is able to synthesize rules purely based on the operators defining the domain. KBC, on the other hand, requires axioms as input. Note that these axioms usually define properties of the operators they contain, but it is nevertheless possible to miss axioms when the domain is complex.

Another distinction lies in how correctness of rules is asserted. `Ruler` shows correctness empirically, while KBC proves each generated rule implicitly. Therefore, the correctness of KBC-generated rules depends entirely on the input axioms. This again requires caution, particularly when dealing with partially defined operators.

Finally, generalizability might be an issue for both methods. `Ruler` derives its rules using EqSat and tries to produce a minimal set. This implies that it is sufficient if any two equivalent terms can be proven equivalent under EqSat. This property does not require confluence. It is therefore not clear whether `Ruler`'s rules are effective outside of EqSat.

KBC-generated rule sets do not have this problem. KBC guarantees confluence in case of successful termination. If termination is not given, confluence may not be assumed. However, by increasing the number of generated rules, the coverage can be increased arbitrarily, as KBC is at least a semi-decision algorithm for equivalence proofs (Dick, 1991), and an optimization problem can be reformulated as an equality proof of the original term and the optimal term.

The generalizability challenge for KBC lies elsewhere. For every domain on which KBC is to be applied, KBO must be defined. In particular, it must be defined in such a way that it aligns with the optimization goal at hand. This is likely not possible for every domain and optimization problem.

6.3 Acyclic E-Graphs

ægraphs, introduced by Fallin (2023), are a special type of e-graph used in Cranelift, the compiler backend of the Wasmtime[1] WebAssembly runtime. The focus of ægraphs is on keeping the e-graph acyclic to allow forward and backward conversion between ægraphs and control flow graphs directly derived from programs, with the goal of enabling the entire optimization process to be performed on ægraphs.

To achieve acyclicity, certain types of rules are disallowed. This includes a strict directionality requirement for most rules. This seems to be in line with the way KBC generates rewrite rules. Both approaches focus on directed rewrites that exclude cyclic derivations. Therefore, KBC-generated rewrite rules might be particularly useful when applied to ægraphs.

[1]`https://wasmtime.dev/`

7 Conclusion

This work evaluated Knuth-Bendix Completion (KBC) as a tool for program optimization by addressing the following research questions:

1. Do KBC-generated rule sets improve equality saturation in terms of simplification effectiveness and running time, compared to handwritten rules?

2. Do KBC-generated rule sets enable greedy rewriting as a viable alternative to equality saturation in scenarios where minimizing running time and memory usage is essential?

To answer these questions, we used KBC to generate different sets of rewrite rules and applied them in equality saturation (EqSat) and greedy rewriting on arithmetic expressions. We also applied EqSat on the same terms using handwritten example rules from a prominent EqSat implementation as a baseline for comparison.

The results revealed that extending the handwritten rule set with KBC-generated rules improves the EqSat process in terms of both simplification speed and effectiveness. For greedy rewriting, we observed that it achieves better running times than EqSat by several orders of magnitude, but is significantly outperformed in terms of simplification effectiveness under the given conditions.

Therefore, we can confirm research question 1, given that the original rules are retained, while we cannot confirm research question 2 based on the results.

Since we restricted the tests conducted in this experiment to a simple setting, allowing for good control over the validity of their results, there are several aspects future work can follow up on.

First, it would be useful to perform similar experiments on different domains, such as bitvectors or Boolean algebra. This would confirm that the findings are generalizable over domains of similar complexity as arithmetic.

It would also be important to perform tests on data generated from real-world applications rather than randomly generated terms. This would better emphasize how common patterns are handled by the different approaches.

Lastly, future work could focus on adapting the KBC algorithm itself to the context of program optimization. When used outside of theorem proving, where formal guarantees of KBC break down by nature of the task, the concept of superposition could be explored more freely as a way to close gaps in term-rewriting systems.

Overall, this work showed that KBC can be a helpful tool in program optimization and represents a first step in exploring systematic rule generation through completion.

Appendix

1 Input Rule Sets

1.1 Original Rule Set from the egg Library's Test Suite (Willsey et al., 2021a)

Listing 1: Base arithmetic rule set from egg's `math.rs`.

```
rw!("comm-add";  "(+ ?a ?b)"        => "(+ ?b ?a)"),
rw!("comm-mul";  "(* ?a ?b)"        => "(* ?b ?a)"),
rw!("assoc-add"; "(+ ?a (+ ?b ?c))" => "(+ (+ ?a ?b) ?c)"),
rw!("assoc-mul"; "(* ?a (* ?b ?c))" => "(* (* ?a ?b) ?c)"),
rw!("sub-canon"; "(- ?a ?b)" => "(+ ?a (* -1 ?b))"),
rw!("div-canon"; "(/ ?a ?b)" => "(* ?a (pow ?b -1))" if
   is_not_zero("?b")),
// rw!("canon-sub"; "(+ ?a (* -1 ?b))"   => "(- ?a ?b)"),
// rw!("canon-div"; "(* ?a (pow ?b -1))" => "(/ ?a ?b)" if
   is_not_zero("?b")),
rw!("zero-add"; "(+ ?a 0)" => "?a"),
rw!("zero-mul"; "(* ?a 0)" => "0"),
rw!("one-mul";  "(* ?a 1)" => "?a"),
rw!("add-zero"; "?a" => "(+ ?a 0)"),
rw!("mul-one";  "?a" => "(* ?a 1)"),
rw!("cancel-sub"; "(- ?a ?a)" => "0"),
rw!("cancel-div"; "(/ ?a ?a)" => "1" if is_not_zero("?a")),
rw!("distribute"; "(* ?a (+ ?b ?c))"        => "(+ (* ?a ?b)
   (* ?a ?c))"),
rw!("factor"    ; "(+ (* ?a ?b) (* ?a ?c))" => "(* ?a (+ ?b ?
   c))"),
rw!("pow-mul"; "(* (pow ?a ?b) (pow ?a ?c))" => "(pow ?a (+ ?
   b ?c))"),
rw!("pow0"; "(pow ?x 0)" => "1"
  if is_not_zero("?x")),
rw!("pow1"; "(pow ?x 1)" => "?x"),
rw!("pow2"; "(pow ?x 2)" => "(* ?x ?x)"),
rw!("pow-recip"; "(pow ?x -1)" => "(/ 1 ?x)"
  if is_not_zero("?x")),
rw!("recip-mul-div"; "(* ?x (/ 1 ?x))" => "1" if is_not_zero("?x")),
rw!("d-variable"; "(d ?x ?x)" => "1" if is_sym("?x")),
rw!("d-constant"; "(d ?x ?c)" => "0" if is_sym("?x") if is_const_or_distinct_var("?c", "?x")),
rw!("d-add"; "(d ?x (+ ?a ?b))" => "(+ (d ?x ?a) (d ?x ?b))"),
rw!("d-mul"; "(d ?x (* ?a ?b))" => "(+ (* ?a (d ?x ?b)) (* ?b (d ?x ?a)))"),
rw!("d-sin"; "(d ?x (sin ?x))" => "(cos ?x)"),
rw!("d-cos"; "(d ?x (cos ?x))" => "(* -1 (sin ?x))"),
rw!("d-ln"; "(d ?x (ln ?x))" => "(/ 1 ?x)" if is_not_zero("?x")),
rw!("d-power";
  "(d ?x (pow ?f ?g))" =>
  "(* (pow ?f ?g)
  (+ (* (d ?x ?f)
  (/ ?g ?f))
  (* (d ?x ?g)
  (ln ?f))))"
  if is_not_zero("?f")
  if is_not_zero("?g")
  ),
rw!("i-one"; "(i 1 ?x)" => "?x"),
rw!("i-power-const"; "(i (pow ?x ?c) ?x)" =>
  "(/ (pow ?x (+ ?c 1)) (+ ?c 1))" if is_const("?c")),
rw!("i-cos"; "(i (cos ?x) ?x)" => "(sin ?x)"),
rw!("i-sin"; "(i (sin ?x) ?x)" => "(* -1 (cos ?x))"),
rw!("i-sum"; "(i (+ ?f ?g) ?x)" => "(+ (i ?f ?x) (i ?g ?x))"),
rw!("i-dif"; "(i (- ?f ?g) ?x)" => "(- (i ?f ?x) (i ?g ?x))"),
rw!("i-parts"; "(i (* ?a ?b) ?x)" =>
  "(- (* ?a (i ?b ?x)) (i (* (d ?x ?a) (i ?b ?x)) ?x))"),
```

1.2 Modified Rule Set math_no_diff_int

Listing 2: Rule set 1.1 without differentiation and integration rules and −1 replaced with neg(1).

```
rw!("comm-add";  "(+ ?x ?y)"        => "(+ ?y ?x)"),
rw!("comm-mul";  "(* ?x ?y)"        => "(* ?y ?x)"),
rw!("assoc-add"; "(+ ?x (+ ?y ?z))" => "(+ (+ ?x ?y) ?z)"),
rw!("assoc-mul"; "(* ?x (* ?y ?z))" => "(* (* ?x ?y) ?z)"),
rw!("sub-canon"; "(- ?x ?y)" => "(+ ?x (* (neg 1) ?y))"),
rw!("div-canon"; "(/ ?x ?y)" => "(* ?x (pow ?y (neg 1)))" if is_not_zero("?y")),
// rw!("canon-sub"; "(+ ?x (* (neg 1) ?y))"   => "(- ?x ?y)"),
// rw!("canon-div"; "(* ?x (pow ?y (neg 1)))" => "(/ ?x ?y)" if is_not_zero("?y")),
rw!("zero-add"; "(+ ?x 0)" => "?x"),
rw!("zero-mul"; "(* ?x 0)" => "0"),
rw!("one-mul";  "(* ?x 1)" => "?x"),
rw!("add-zero"; "?x" => "(+ ?x 0)"),
rw!("mul-one";  "?x" => "(* ?x 1)"),
rw!("cancel-sub"; "(- ?x ?x)" => "0"),
rw!("cancel-div"; "(/ ?x ?x)" => "1" if is_not_zero("?x")),
rw!("distribute"; "(* ?x (+ ?y ?z))"        => "(+ (* ?x ?y) (* ?x ?z))"),
rw!("factor"    ; "(+ (* ?x ?y) (* ?x ?z))" => "(* ?x (+ ?y ?z))"),
rw!("pow-mul"; "(* (pow ?x ?y) (pow ?x ?z))" => "(pow ?x (+ ?y ?z))" if is_not_zero("?x")),
rw!("pow0"; "(pow ?x 0)" => "1" if is_not_zero("?x")),
rw!("pow1"; "(pow ?x 1)" => "?x"),
rw!("pow2"; "(pow ?x 2)" => "(* ?x ?x)"),
rw!("pow-recip"; "(pow ?x (neg 1))" => "(/ 1 ?x)" if is_not_zero("?x")),
rw!("recip-mul-div"; "(* ?x (/ 1 ?x))" => "1" if is_not_zero("?x")),
```

1.3 Modified Rule Set math_no_diff_int_no_div_no_pow

Listing 3: Rule set 1.2 without rules containing division and power operators.

```
rw!("comm-add";  "(+ ?x ?y)"         => "(+ ?y ?x)"),
rw!("comm-mul";  "(* ?x ?y)"         => "(* ?y ?x)"),
rw!("assoc-add"; "(+ ?x (+ ?y ?z))" => "(+ (+ ?x ?y) ?z)"),
rw!("assoc-mul"; "(* ?x (* ?y ?z))" => "(* (* ?x ?y) ?z)"),
rw!("sub-canon"; "(- ?x ?y)" => "(+ ?x (* (neg 1) ?y))"),
// rw!("canon-sub"; "(+ ?x (* (neg 1) ?y))"   => "(- ?x ?y)")
   ,
// rw!("canon-div"; "(* ?x (pow ?y (neg 1)))" => "(/ ?x ?y)"
   if is_not_zero("?y")),
rw!("zero-add"; "(+ ?x 0)" => "?x"),
rw!("zero-mul"; "(* ?x 0)" => "0"),
rw!("one-mul";  "(* ?x 1)" => "?x"),
rw!("add-zero"; "?x" => "(+ ?x 0)"),
rw!("mul-one";  "?x" => "(* ?x 1)"),
rw!("cancel-sub"; "(- ?x ?x)" => "0"),
rw!("distribute"; "(* ?x (+ ?y ?z))"        => "(+ (* ?x ?y)
   (* ?x ?z))"),
rw!("factor"    ; "(+ (* ?x ?y) (* ?x ?z))" => "(* ?x (+ ?y ?
   z))"),
```

1.4 Modified Rule Set math_no_diff_int_sep_div

Listing 4: Rule set 1.2 with seperately completed rules for division.

```
rw!("comm-add";  "(+ ?x ?y)"        => "(+ ?y ?x)"),
rw!("comm-mul";  "(* ?x ?y)"        => "(* ?y ?x)"),
rw!("assoc-add"; "(+ ?x (+ ?y ?z))" => "(+ (+ ?x ?y) ?z)"),
rw!("assoc-mul"; "(* ?x (* ?y ?z))" => "(* (* ?x ?y) ?z)"),
rw!("sub-canon"; "(- ?x ?y)" => "(+ ?x (* (neg 1) ?y))"),
rw!("div-canon"; "(/ ?x ?y)" => "(* ?x (pow ?y (neg 1)))" if is_not_zero("?y")),
// rw!("canon-sub"; "(+ ?x (* (neg 1) ?y))"   => "(- ?x ?y)"),
// rw!("canon-div"; "(* ?x (pow ?y (neg 1)))" => "(/ ?x ?y)" if is_not_zero("?y")),
rw!("zero-add"; "(+ ?x 0)" => "?x"),
rw!("zero-mul"; "(* ?x 0)" => "0"),
rw!("one-mul";  "(* ?x 1)" => "?x"),
rw!("add-zero"; "?x" => "(+ ?x 0)"),
rw!("mul-one";  "?x" => "(* ?x 1)"),
rw!("cancel-sub"; "(- ?x ?x)" => "0"),
rw!("cancel-div"; "(/ ?x ?x)" => "1" if is_not_zero("?x")),
rw!("distribute"; "(* ?x (+ ?y ?z))"        => "(+ (* ?x ?y) (* ?x ?z))"),
rw!("factor"    ; "(+ (* ?x ?y) (* ?x ?z))" => "(* ?x (+ ?y ?z))"),
rw!("pow-mul"; "(* (pow ?x ?y) (pow ?x ?z))" => "(pow ?x (+ ?y ?z))") if is_not_zero("?x"),
rw!("pow0"; "(pow ?x 0)" => "1" if is_not_zero("?x")),
rw!("pow1"; "(pow ?x 1)" => "?x"),
rw!("pow2"; "(pow ?x 2)" => "(* ?x ?x)"),
rw!("pow-recip"; "(pow ?x (neg 1))" => "(/ 1 ?x)" if is_not_zero("?x")),
rw!("recip-mul-div"; "(* ?x (/ 1 ?x))" => "1" if is_not_zero("?x")),
rw!("guarded_rule_0"; "(* ?x ?x)" => "(pow ?x 2)"),
rw!("guarded_rule_1"; "(* ?x ?y)" => "(* ?y ?x)"),
rw!("guarded_rule_2"; "(* ?x 1)" => "?x"),
rw!("guarded_rule_3"; "(* 1 ?x)" => "?x"),
rw!("guarded_rule_4"; "(/ ?x ?x)" => "1" if is_not_zero("?x")),
rw!("guarded_rule_5"; "(/ ?x 1)" => "?x"),
rw!("guarded_rule_6"; "(pow ?x 1)" => "?x"),
rw!("guarded_rule_7"; "(pow 1 2)" => "1"),
rw!("guarded_rule_8"; "(pow ?x (neg 1) )" => "(/ 1 ?x)" if is_not_zero("?x")),
rw!("guarded_rule_9"; "(* ?x (* ?x ?y) )" => "(* (pow ?x 2) ?y)"),
rw!("guarded_rule_10"; "(* ?x (/ ?y ?z) )" => "(/ (* ?x ?y) ?z)" if is_not_zero("?z")),
rw!("guarded_rule_11"; "(* ?x (pow ?x ?y) )" => "(pow ?x (+ ?y 1) )" if is_not_zero("?x")),
rw!("guarded_rule_12"; "(* ?y (* ?y ?x) )" => "(* ?x (pow ?y 2) )"),
rw!("guarded_rule_13"; "(* ?x (* ?y ?z) )" => "(* ?y (* ?x ?z) )"),
rw!("guarded_rule_14"; "(* (* ?x ?y) ?z)" => "(* ?x (* ?y ?z) )"),
rw!("guarded_rule_15"; "(* (/ ?x ?y) ?z)" => "(/ (* ?x ?z) ?y)" if is_not_zero("?y")),
rw!("guarded_rule_16"; "(/ ?x (/ ?x ?y) )" => "?y" if is_not_zero("?y") if is_not_zero("?x")),
rw!("guarded_rule_17"; "(/ (* ?x ?y) ?y)" => "?x" if is_not_zero("?y")),
rw!("guarded_rule_18"; "(/ (/ ?x ?y) ?x)" => "(/ 1 ?y)" if is_not_zero("?x") if is_not_zero("?y")),
rw!("guarded_rule_19"; "(/ (/ ?x ?y) ?z)" => "(/ (/ ?x ?z) ?y)" if is_not_zero("?z") if is_not_zero("?y")),
rw!("guarded_rule_20"; "(/ (/ 1 ?x) ?x)" => "(pow (/ 1 ?x) 2)" if is_not_zero("?x")),
rw!("guarded_rule_21"; "(/ (pow ?x 2) ?x)" => "?x" if is_not_zero("?x")),
rw!("guarded_rule_22"; "(pow ?x (+ ?y ?y) )" => "(pow (pow ?x ?y) 2)" if is_not_zero("?x")),
rw!("guarded_rule_23"; "(pow ?x (+ 1 1) )" => "(pow ?x 2)"),
rw!("guarded_rule_24"; "(pow ?x (+ ?y ?z) )" => "(pow ?x (+ ?z ?y) )"),
rw!("guarded_rule_25"; "(pow 1 (+ ?x 1) )" => "(pow 1 ?x)"),
rw!("guarded_rule_26"; "(pow 1 (+ ?x 2) )" => "(pow 1 ?x)"),
rw!("guarded_rule_27"; "(* (pow ?x ?y) (pow ?x ?z) )" => "(pow ?x (+ ?y ?z) )" if is_not_zero("?x")),
```

1.5 Modified Rule Set math_no_diff_int_sep_div_plus

Listing 5: Rule set 1.4 with seperately completed rules for division, including intermediate rules deleted during completion.

```
rw!("comm-add";  "(+ ?x ?y)"        => "(+ ?y ?x)"),
rw!("comm-mul";  "(* ?x ?y)"        => "(* ?y ?x)"),
rw!("assoc-add"; "(+ ?x (+ ?y ?z))" => "(+ (+ ?x ?y) ?z)"),
rw!("assoc-mul"; "(* ?x (* ?y ?z))" => "(* (* ?x ?y) ?z)"),
rw!("sub-canon"; "(- ?x ?y)" => "(+ ?x (* (neg 1) ?y))"),
rw!("div-canon"; "(/ ?x ?y)" => "(* ?x (pow ?y (neg 1)))" if is_not_zero("?y")),
// rw!("canon-sub"; "(+ ?x (* (neg 1) ?y))"   => "(- ?x ?y)"),
// rw!("canon-div"; "(* ?x (pow ?y (neg 1)))" => "(/ ?x ?y)" if is_not_zero("?y")),
rw!("zero-add"; "(+ ?x 0)" => "?x"),
rw!("zero-mul"; "(* ?x 0)" => "0"),
rw!("one-mul";  "(* ?x 1)" => "?x"),
rw!("add-zero"; "?x" => "(+ ?x 0)"),
rw!("mul-one";  "?x" => "(* ?x 1)"),
rw!("cancel-sub"; "(- ?x ?x)" => "0"),
rw!("cancel-div"; "(/ ?x ?x)" => "1" if is_not_zero("?x")),
rw!("distribute"; "(* ?x (+ ?y ?z))"        => "(+ (* ?x ?y) (* ?x ?z))"),
rw!("factor"    ; "(+ (* ?x ?y) (* ?x ?z))" => "(* ?x (+ ?y ?z))"),
rw!("pow-mul"; "(* (pow ?x ?y) (pow ?x ?z))" => "(pow ?x (+ ?y ?z))") if is_not_zero("?x"),
rw!("pow0"; "(pow ?x 0)" => "1" if is_not_zero("?x")),
rw!("pow1"; "(pow ?x 1)" => "?x"),
rw!("pow2"; "(pow ?x 2)" => "(* ?x ?x)"),
rw!("pow-recip"; "(pow ?x (neg 1))" => "(/ 1 ?x)" if is_not_zero("?x")),
rw!("recip-mul-div"; "(* ?x (/ 1 ?x))" => "1" if is_not_zero("?x")),
rw!("guarded_rule_0"; "(/ ?x ?x)" => "1" if is_not_zero("?x")),
rw!("guarded_rule_1"; "(pow ?x 1)" => "?x"),
rw!("guarded_rule_2"; "(pow ?x (neg 1))" => "(/ 1 ?x)" if is_not_zero("?x")),
rw!("guarded_rule_3"; "(* ?x ?x)" => "(pow ?x 2)"),
rw!("guarded_rule_4"; "(* ?x ?y)" => "(* ?y ?x)"),
rw!("guarded_rule_5"; "(* ?x 1)" => "?x"),
rw!("guarded_rule_6"; "(* ?x (/ 1 ?x) )" => "1" if
    is_not_zero("?x")),
rw!("guarded_rule_7"; "(* ?x (/ 1 ?y) )" => "(/ ?x ?y)" if
    is_not_zero("?y")),
rw!("guarded_rule_8"; "(* (* ?x ?y) ?z)" => "(* ?x (* ?y ?z)
    )"),
rw!("guarded_rule_9"; "(* (pow ?x ?y) (pow ?x ?z) )" => "(pow
     ?x (+ ?y ?z) )" if is_not_zero("?x")),
rw!("guarded_rule_10"; "(* 1 ?x)" => "?x"),
rw!("guarded_rule_11"; "(/ ?x 1)" => "?x"),
rw!("guarded_rule_12"; "(pow 1 2)" => "1"),
rw!("guarded_rule_13"; "(* (/ 1 ?x) ?y)" => "(/ ?y ?x)" if
    is_not_zero("?x")),
rw!("guarded_rule_14"; "(* ?x (* ?y ?z) )" => "(* ?y (* ?x ?z
    ) )"),
rw!("guarded_rule_15"; "(* ?x (/ ?y ?x) )" => "?y" if
    is_not_zero("?x")),
rw!("guarded_rule_16"; "(* ?x (/ ?y ?z) )" => "(/ (* ?x ?y) ?
    z)" if is_not_zero("?z")),
rw!("guarded_rule_17"; "(/ (* ?x ?y) ?x)" => "?y" if
    is_not_zero("?x")),
rw!("guarded_rule_18"; "(pow ?x (+ ?y ?z) )" => "(pow ?x (+ ?
    z ?y) )"),
rw!("guarded_rule_20"; "(/ (pow ?x 2) ?x)" => "?x" if
    is_not_zero("?x")),
rw!("guarded_rule_21"; "(* ?x (pow ?x ?y) )" => "(pow ?x (+ ?
    y 1) )" if is_not_zero("?x")),
rw!("guarded_rule_22"; "(pow 1 (+ 2 1) )" => "1"),
rw!("guarded_rule_24"; "(pow 1 (+ 2 2) )" => "1"),
rw!("guarded_rule_25"; "(/ (* ?x ?y) ?y)" => "?x" if
    is_not_zero("?y")),
rw!("guarded_rule_27"; "(pow 1 (+ ?x 2) )" => "(pow 1 ?x)"),
rw!("guarded_rule_28"; "(pow ?x (+ 1 1) )" => "(pow ?x 2)"),
rw!("guarded_rule_29"; "(pow 1 (+ ?x 1) )" => "(pow 1 ?x)"),
rw!("guarded_rule_30"; "(* (/ ?x ?y) ?z)" => "(/ (* ?x ?z) ?y
    )" if is_not_zero("?y")),
rw!("guarded_rule_31"; "(/ (/ ?x ?y) ?z)" => "(/ (/ ?x ?z) ?y
    )" if is_not_zero("?y") if is_not_zero("?z")),
rw!("guarded_rule_33"; "(/ (/ ?x ?y) ?x)" => "(/ 1 ?y)" if
    is_not_zero("?y") if is_not_zero("?x")),
rw!("guarded_rule_35"; "(/ ?x (/ ?x ?y) )" => "?y" if
    is_not_zero("?x") if is_not_zero("?y")),
```

2 Test Data Tables

Table 1: EqSat simplification effectiveness by postprocessing on test sets with 25 binary operators. Higher values indicate more effective simplification. Data for Fig. 4.1.

Configuration	Test Set	0.0001 s	0.0005 s	0.001 s	0.005 s	0.01 s	0.05 s	0.1 s	0.5 s	1 s
EqSat_KBC_150_extended	random_terms_huge	14.344	15.776	21.120	24.040	24.400	24.728	24.744	24.832	24.832
EqSat_KBC_150_extended_no_unorderable	random_terms_huge	14.864	21.088	22.040	24.376	24.608	24.736	24.768	24.816	24.832
EqSat_KBC_150	random_terms_huge	14.192	19.232	19.888	20.152	20.168	20.192	20.192	20.200	20.200
EqSat_KBC_150_no_unorderable	random_terms_huge	15.928	19.824	19.968	20.072	20.088	20.088	20.112	20.120	20.120
EqSat_no_div_no_pow_KBC_100_extended	no_div_no_pow_random_terms_huge	18.664	21.232	26.768	30.472	30.840	32.536	32.632	33.064	33.208
EqSat_no_div_no_pow_KBC_100_extended_no_unorderable	no_div_no_pow_random_terms_huge	18.952	26.448	26.880	30.584	31.600	32.488	32.720	33.056	33.248
EqSat_no_div_no_pow_KBC_100	no_div_no_pow_random_terms_huge	18.024	26.032	28.752	31.152	31.688	32.608	32.840	33.128	33.184
EqSat_no_div_no_pow_KBC_100_no_unorderable	no_div_no_pow_random_terms_huge	18.536	28.192	29.368	31.168	31.488	32.064	32.128	32.288	32.352

Table 2: EqSat simplification effectiveness by rule set size on test sets with 25 binary operators. Higher values indicate more effective simplification. Data for Fig. 4.2.

Configuration	Test Set	0.0001 s	0.0005 s	0.001 s	0.005 s	0.01 s	0.05 s	0.1 s	0.5 s	1 s
EqSat_KBC_100	random_terms_huge	14.520	19.592	19.840	19.936	19.936	19.944	19.944	19.944	19.944
EqSat_KBC_150	random_terms_huge	14.192	19.232	19.888	20.152	20.168	20.192	20.192	20.200	20.200
EqSat_KBC_200	random_terms_huge	11.000	18.800	19.608	20.104	20.176	20.200	20.216	20.216	20.216
EqSat_KBC_40	random_terms_huge	18.032	18.920	18.952	18.952	18.952	18.952	18.952	18.952	18.952
EqSat_KBC_100_extended	random_terms_huge	14.888	21.016	21.744	24.360	24.600	24.752	24.784	24.808	24.816
EqSat_KBC_150_extended	random_terms_huge	14.344	15.776	21.120	24.040	24.400	24.728	24.744	24.832	24.832
EqSat_KBC_200_extended	random_terms_huge	8.928	14.904	20.480	23.320	24.112	24.624	24.696	24.800	24.832
EqSat_KBC_40_extended	random_terms_huge	14.832	21.272	23.184	24.464	24.592	24.744	24.784	24.816	24.816

Table 3: EqSat simplification effectiveness for small terms (1–3 binary operators). Higher values indicate more effective simplification. Data for Figure 4.3.

Configuration	Test Set	0.0001 s	0.0005 s	0.001 s	0.005 s	0.01 s	0.05 s	0.1 s	0.5 s	1 s
EqSat_no_div_no_pow	no_div_no_pow_random_terms_small	3.322	3.664	3.668	3.668	3.668	3.668	3.668	3.668	3.668
EqSat_no_div_no_pow_KBC_100_extended	no_div_no_pow_random_terms_small	3.256	3.772	3.828	3.856	3.856	3.856	3.856	3.856	3.856
EqSat_no_div_no_pow_KBC_100_extended_no_unorderable	no_div_no_pow_random_terms_small	3.464	3.826	3.854	3.856	3.856	3.856	3.856	3.856	3.856
EqSat_no_div_no_pow_KBC_100	no_div_no_pow_random_terms_small	3.650	3.814	3.816	3.816	3.816	3.816	3.816	3.816	3.816
EqSat_no_div_no_pow_KBC_100_no_unorderable	no_div_no_pow_random_terms_small	3.738	3.762	3.762	3.762	3.762	3.762	3.762	3.762	3.762
EqSat_base	random_terms_small	2.632	2.810	2.810	2.810	2.810	2.810	2.810	2.810	2.810
EqSat_KBC_150_extended_no_unorderable	random_terms_small	2.636	2.870	2.902	2.908	2.908	2.908	2.908	2.908	2.908
EqSat_sep_div_KBC_150_extended_no_unorderable	random_terms_small	2.756	2.926	2.936	2.940	2.940	2.940	2.940	2.940	2.940
EqSat_sep_div_plus_KBC_150_extended_no_unorderable	random_terms_small	2.776	2.932	2.938	2.940	2.940	2.940	2.940	2.940	2.940

Table 4: EqSat simplification effectiveness for large terms (10 binary operators). Higher values indicate more effective simplification. Data for Figure 4.4.

Configuration	Test Set	0.0001 s	0.0005 s	0.001 s	0.005 s	0.01 s	0.05 s	0.1 s	0.5 s	1 s
EqSat_no_div_no_pow	no_div_no_pow_random_terms_large	9.536	11.772	12.456	13.376	13.508	13.572	13.588	13.608	13.608
EqSat_no_div_no_pow_KBC_100_extended	no_div_no_pow_random_terms_large	8.824	11.428	12.160	13.688	13.972	14.100	14.108	14.132	14.132
EqSat_no_div_no_pow_KBC_100_extended_no_unorderable	no_div_no_pow_random_terms_large	8.816	11.856	13.128	13.860	13.996	14.092	14.104	14.128	14.132
EqSat_no_div_no_pow_KBC_100	no_div_no_pow_random_terms_large	9.308	12.644	13.392	13.960	14.032	14.104	14.108	14.120	14.124
EqSat_no_div_no_pow_KBC_100_no_unorderable	no_div_no_pow_random_terms_large	10.872	13.028	13.392	13.680	13.720	13.744	13.752	13.764	13.764
EqSat_base	random_terms_large	7.332	9.372	9.584	9.728	9.728	9.728	9.728	9.728	9.728
EqSat_KBC_150_extended_no_unorderable	random_terms_large	6.852	9.464	9.940	10.284	10.300	10.324	10.324	10.324	10.324
EqSat_sep_div_KBC_150_extended_no_unorderable	random_terms_large	7.280	9.756	10.196	10.448	10.452	10.460	10.460	10.460	10.460
EqSat_sep_div_plus_KBC_150_extended_no_unorderable	random_terms_large	8.144	9.816	10.224	10.448	10.452	10.460	10.460	10.460	10.460

Table 5: EqSat simplification effectiveness for huge terms (25 binary operators). Higher values indicate more effective simplification. Data underlying Figure 4.5.

Configuration	**Test Set**	**0.0001 s**	**0.0005 s**	**0.001 s**	**0.005 s**	**0.01 s**	**0.05 s**	**0.1 s**	**0.5 s**	**1 s**
EqSat_no_div_no_pow	no_div_no_pow_random_terms_huge	16.008	25.888	26.984	29.488	30.224	31.184	31.416	31.800	31.824
EqSat_no_div_no_pow_KBC_100_extended	no_div_no_pow_random_terms_huge	18.664	21.232	26.768	30.472	30.840	32.536	32.632	33.064	33.208
EqSat_no_div_no_pow_KBC_100_extended_no_unorderable	no_div_no_pow_random_terms_huge	18.952	26.448	26.880	30.584	31.600	32.488	32.720	33.056	33.248
EqSat_no_div_no_pow_KBC_100	no_div_no_pow_random_terms_huge	18.024	26.032	28.752	31.152	31.688	32.608	32.840	33.128	33.184
EqSat_no_div_no_pow_KBC_100_no_unorderable	no_div_no_pow_random_terms_huge	18.536	28.192	29.368	31.168	31.488	32.064	32.128	32.288	32.352
EqSat_base	random_terms_huge	13.560	21.416	22.512	23.240	23.376	23.408	23.424	23.448	23.448
EqSat_KBC_150_extended_no_unorderable	random_terms_huge	14.864	21.088	22.040	24.376	24.608	24.736	24.768	24.816	24.832
EqSat_sep_div_KBC_150_extended_no_unorderable	random_terms_huge	15.752	22.256	22.784	24.784	24.936	25.080	25.080	25.088	25.104
EqSat_sep_div_plus_KBC_150_extended_no_unorderable	random_terms_huge	17.056	22.728	23.232	24.808	24.912	25.056	25.056	25.064	–

Table 6: Greedy rewriting simplification effectiveness for huge terms (25 binary operators). Higher values indicate more effective simplification. Data for Figure 4.6.

Configuration	**Test Set**	**0.0001 s**	**0.0005 s**	**0.001 s**	**0.005 s**	**0.01 s**	**0.05 s**	**0.1 s**	**0.5 s**
greedy_no_div_no_pow_KBC	no_div_no_pow_random_terms_huge	22.712	24.240	24.360	24.448	24.840	25.000	25.376	25.432
greedy_no_div_no_pow_KBC_no_unorderable	no_div_no_pow_random_terms_huge	22.696	24.264	24.440	24.496	24.880	25.104	25.360	25.408
greedy_no_div_no_pow	no_div_no_pow_random_terms_huge	17.472	17.472	17.472	17.472	17.472	17.472	17.472	17.472
greedy_KBC	random_terms_huge	17.136	17.304	17.272	17.496	17.752	17.768	17.816	17.808
greedy_KBC_no_unorderable	random_terms_huge	17.136	17.304	17.272	17.456	17.456	17.720	17.784	17.816
greedy_base	random_terms_huge	14.472	14.472	14.472	14.472	14.472	14.472	14.472	14.472

3 EqSat Output Examples

Table 7: Example EqSat simplification results for individual terms. Simplification time was provided directly by `egg` (Willsey et al., 2021b).

Configuration	**Input**	**Simplified**	**L**	**S**	**I**
EqSat_base	`(- a (+ (pow a 1) a))`	`(* a -1)`	0.01	0.0281	11
EqSat_base	`(* a (+ (- -1 a) a))`	`(* a -1)`	0.10	0.3803	9
EqSat_base	`(+ a (- (pow a -1) a))`	`(/ 1 a)`	1.00	3.2423	14

Legend: L = Time limit [s]; S = Simplification time [s]; I = Number of iterations performed by EqSat. The `Input` and `Simplified` columns show terms in `egg`-compatible syntax.

4 Declaration of the Usage of Generative AI

Generative AI, namely ChatGPT and Github Copilot were used during the writing of this thesis. Both tools were used in the conventional manner and scope to develop and edit source code and written text.

Source code in which generative AI produced any logic is marked accordingly with code comments.

Bibliography

Baader, F. and Nipkow, T. (1998). *Term Rewriting and All That.* Cambridge University Press.

Cai, Y., Yang, K., Deng, C., Yu, C., and Zhang, Z. (2025). Smoothe: Differentiable e-graph extraction. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 1020–1034, New York, NY, USA. Association for Computing Machinery.

Claessen, K. and Smallbone, N. (2021). Efficient encodings of first-order horn formulas in equational logic. In *Proceedings of the Twenty-Fourth International Conference on Automated Reasoning (CADE-24).* Accessed: 2025-10-21.

Dick, A. J. J. (1991). An introduction to knuth-bendix completion. *Comput. J.*, 34(1):2–15.

Fallin, C. (2023). ægraphs: Acyclic e-graphs for efficient optimization in a production compiler. `https://cfallin.org/pubs/egraphs2023_aegraphs_slides.pdf`. Presentation slides, November 2023.

Hong, C. (2024). A survey of rewrite rule synthesis. `https://inst.eecs.berkeley.edu/~cs294-260/sp24/projects/charleshong/`. Course project report, University of California, Berkeley.

Nandi, C., Willsey, M., Zhu, A., Wang, Y. R., Saiki, B., Anderson, A., Schulz, A., Grossman, D., and Tatlock, Z. (2021). Rewrite rule inference using equality saturation. *Proc. ACM Program. Lang.*, 5(OOPSLA).

Panchekha, P., Sanchez-Stern, A., Wilcox, J. R., and Tatlock, Z. (2015). Automatically improving accuracy for floating point expressions. *SIGPLAN Not.*, 50(6):1–11.

Smallbone, N. (2021). Twee: An equational theorem prover. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*, page 602–613, Berlin, Heidelberg. Springer-Verlag.

Tate, R., Stepp, M., Tatlock, Z., and Lerner, S. (2011). Equality saturation: A new approach to optimization. *Logical Methods in Computer Science*, Volume 7, Issue 1.

Willsey, M., Nandi, C., Wang, Y. R., Flatt, O., Tatlock, Z., and Panchekha, P. (2021a). egg: e-graphs good! — math rewrite rules. `https://github.com/egraphs-good/egg/blob/main/tests/math.rs`. Accessed: 2025-10-22.

Willsey, M., Nandi, C., Wang, Y. R., Flatt, O., Tatlock, Z., and Panchekha, P. (2021b). egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29.

Yin, J., Song, Z., Chen, C., Cai, Y., Zhang, Z., and Yu, C. (2025). e-boost: Boosted e-graph extraction with adaptive heuristics and exact solving.

Zhang, Y. and Flatt, O. (2023). Ensuring the termination of equality saturation for terminating term rewriting systems. Accessed: 2025-10-17.

Zhang, Y., Wang, Y. R., Flatt, O., Cao, D., Zucker, P., Rosenthal, E., Tatlock, Z., and Willsey, M. (2023). Better together: Unifying datalog and equality saturation.

Zhang, Y., Wang, Y. R., Willsey, M., and Tatlock, Z. (2022). Relational e-matching. *Proc. ACM Program. Lang.*, 6(POPL).

Zhao, Y., Sharif, H., Adve, V., and Misailovic, S. (2024). Felix: Optimizing tensor programs with gradient descent. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 367–381, New York, NY, USA. Association for Computing Machinery.