Conditional Rewrite Rule Synthesis with E-Graphs and Implication Propagation

Andrew Cheung
UC San Diego
San Diego, USA
a7cheung@ucsd.edu

Chandrakana Nandi
Certora Inc.
Seattle, USA
chandra@certora.com

Sorin Lerner
Cornell University
Ithaca, USA
sorin.lerner@cornell.edu

***Abstract*—Compilers, synthesizers, and theorem provers rely on rich rulesets for manipulating expressions via rewriting, but developing such rules remains difficult and error-prone. While prior work has automated synthesis of rewrite rules, it has focused largely on strict equalities, leaving conditional rewrites largely unsupported. Conditional rule synthesis explores a much larger search space because it must search for both the rule body and the condition. Effective pruning of the space is crucial for scaling synthesis to large grammars. The commonly used "derivability" metric for eliminating redundant rules in the strict setting does not extend directly to the conditional case. A conditional rule may be redundant not only when its body and condition are equivalent to a rule, but also when its condition *implies* a condition of a more general rule. We present CHOMPY, a tool for synthesizing conditional rewrite rules. CHOMPY's core contribution is *implication propagation*, a technique that lifts equality saturation to the predicate level to discover logical implications between conditions, then uses those implications to obtain a new notion of conditional derivability and ruleset minimization. To evaluate CHOMPY, we compare the synthesized conditional ruleset to handwritten ones from Caviar (an equality saturation theorem prover) and Halide's term rewriting system. CHOMPY's rewrites derive 73.3% of Caviar's and 57.1% of Halide's conditional rules. CHOMPY outperforms an LLM-only baseline on both benchmarks. Unlike the LLM's substantially variable output across runs, CHOMPY's results are deterministic and stable.**

I. INTRODUCTION

Rewrite rules are a fundamental component of compilers, theorem provers, and program synthesizers where they are used to transform programs to equivalent variants. The process of coming up with rewrite rules is, however, error-prone and tedious. Rule synthesis tools and theory explorers automatically discover such rewrite rules. Recent work has shown how equality saturation can be used to synthesize rewrite rules for large, complex grammars [1], [2] and perform theory exploration [3], [4], [5], but most of the existing work with equality saturation has focused mainly on total rules. Conditional rewrite rules (e.g., $\texttt{if } x \neq 0 \texttt{ then } x/x \rightsquigarrow 1$) are key to many applications but have not been the focus of much prior work on automated rewrite rule synthesis.

Extending theory explorers to support conditional rules presents a key challenge. The introduction of condition terms greatly expands the search space and without effective filtering, can lead to the generation of too many unimportant rules.

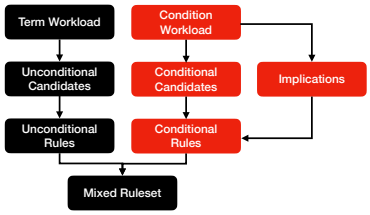

Fig. 1: CHOMPY extends traditional equality saturation-based theory explorers (black) for conditional rule synthesis (red).

This can hinder the performance of the underlying rewrite systems. A conditional rule can be filtered out not only when both the rule's condition and body are equivalent to those of another rule, but also when the rule's condition implies a more general one. Many rewrite rule candidates differ only in the strength of their side conditions, and traditional techniques for ruleset minimization cannot recognize one as redundant when its condition implies another. Naively comparing conditions pairwise with an SMT solver scales quadratically in the size of the condition workload and quickly becomes infeasible.

In this paper, we present **CHOMPY**, an equality saturation-based rewrite rule synthesis tool that synthesizes both total and conditional rewrite rules. CHOMPY extends prior work [1], [2] on rewrite rule synthesis. The core technical contribution of CHOMPY is **implication propagation**: a new technique that lifts the same equality saturation machinery used for term synthesis to the level of *predicates*, discovering implications between side conditions and using them to obtain *conditional derivability* and *conditional ruleset minimization*. To our knowledge, this is the first systematic answer to conditional minimization in equality-saturation-based rule synthesis.

We present an early evaluation of CHOMPY on a Halide-inspired grammar against two handwritten rulesets: Caviar [6] and the Halide term rewriting system [7]. CHOMPY's rules derive 73.3% of Caviar's conditional rules and 57.1% of Halide's. We also prompted an LLM to synthesize rules for the same grammar and found that compared to CHOMPY, the LLM rulesets are not as effective and also suffer from huge variance across runs. An ablation shows that implication propagation is

https://doi.org/10.34727/2026/isbn.978-3-85448-093-8_43

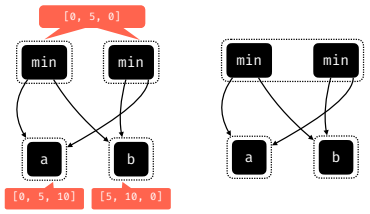

Fig. 2: (Black) An e-graph before and after merging `min(a, b)` and `min(b, a)`. Solid boxes are e-nodes; dotted boxes are e-classes. (Orange) cvecs annotating each e-class. Variables `a`, `b` get base cvecs (often randomly generated and including important constants), and an interpreter computes cvecs for other terms. Note that Figure 3 uses different cvecs for the variable `a` for ease of exposition.

crucial for conditional rule synthesis in CHOMPY.

II. BACKGROUND

E-graphs and equality saturation. E-graphs compactly represent many equivalent terms by sharing subterms: an *e-node* represents a term, an *e-class* groups equivalent ones. Rewrite rules ($l \rightsquigarrow r$) are applied via *ematching* [8] to union e-classes. Figure 2 (right) shows the effect of applying `min(a, b)` $\rightsquigarrow$ `min(b, a)` [1]. Once rewriting reaches saturation or a resource limit, a canonical term is *extracted* from each e-class [9], [10]; CHOMPY extracts by AST size.

Rewrite rule synthesis. Recent work uses equality saturation for fast rewrite rule synthesis [1], [2]. These tools take a term grammar G, an interpreter Φ, and a validator, and proceed in three stages: enumerate terms, identify candidates, minimize. We describe the algorithm with a running example over operators `+`, `min`, `>`, `<`, and variables `a`, `b`; CHOMPY's contribution (Section III) extends this pipeline (Phase 2 in Algorithm 1).

Enumeration (Line 2). Naive enumeration of terms from G scales poorly. Prior work [1] introduces a DSL, ENUMO, in which users write programs to materialize *workloads* from G (filtering operators, plugging terms into terms, unioning workloads, etc.). Each enumerated term is added to a term e-graph E and annotated with a *characteristic vector* (cvec) [2], computed by using the given interpreter to evaluate the term on concrete inputs. The orange portions of Figure 2 show cvecs.

Candidate identification (Line 3). *Cvec matching* [1], [2], [11] produces candidates by searching E for distinct e-classes with equal cvecs: for e_1, e_2 with $c_1 = c_2$, the candidate $l \rightsquigarrow r$ (canonical terms of e_1, e_2) is added to a set of total rewrite candidates C if a validator (e.g. Z3) certifies it. In our running example, this finds `min(a, b)` $\rightsquigarrow$ `min(b, a)`.

Minimization (Line 4). Candidates are filtered to a non-redundant set R as described via the black-text portions of Algorithm 2. A "best rule" is repeatedly popped from C by a syntactic heuristic (e.g. smallest size) and added to R.

[1]For readability, we do not distinguish between pattern variables and program variables; in practice, pattern variables are represented with a `?` prefix.

Algorithm 1: CHOMPY's rule synthesis algorithm, extending prior work [1], [2]. CHOMPY's additions are in orange.

Input: term workload W, conditional workload W_c, interpreter Φ
Output: ruleset R

1 // Phase 1: nonconditional rules
2 $E \leftarrow$ NEWEGRAPH(W, Φ)
3 $C \leftarrow$ CVECMATCH(E, Φ)
4 $R \leftarrow$ MINIMIZE$([], C, [], \Phi)$
5 // Phase 2: Chompy's contribution
6 $E \leftarrow$ COMPRESS(R, E)
7 $E_c \leftarrow$ NEWEGRAPH(W_c, Φ)
8 $P \leftarrow$ GETPVECS(E_c)
9 $I \leftarrow$ FINDIMPLICATIONS(E_c)
10 $C_c \leftarrow$ CONDCVECMATCH(E, P)
11 $R \leftarrow$ MINIMIZE(R, C_c, I, Φ)
12 **return** R

Remaining candidates whose left- and right-hand sides merge under R are discarded. Section III extends this to conditional rules.

III. CORE TECHNIQUE

This section describes how CHOMPY extends rewrite rule synthesis to conditional equalities (Figure 1's red portion). In Algorithm 1, line 6–line 11 extend the pipeline for total rewrite rule synthesis to synthesize conditional rules. We continue the running example from Section II, where CHOMPY discovered total rules over a small grammar consisting of the variables `a`, `b` and the operators `min`, `+`, `>`, `<`. To discover conditional rules, CHOMPY requires two artifacts from Phase 1: the e-graph of terms (E), and the rules generated so far (R). Phase 2 begins with a "compression phase" of running R on E, causing E to union relevant e-classes (Line 6). For the purposes of our running example, R contains the rules `min(a, b)` $\rightsquigarrow$ `min(b, a)`, and `a < b` $\rightsquigarrow$ `b > a`.

A. Enumeration (Lines 7-8)

In Phase 2, enumeration begins from the *condition* workload W_c. For this example, we start with a workload of terms consisting of the operators `<`, `<=`, `>`, `>=`, the variables `a`, `b`, and the constants 0, -2. As in Phase 1, CHOMPY constructs an e-graph E_c from W_c (Line 7). As each condition is added to E_c, its e-class is annotated with a vector of Boolean values. To distinguish these from the arithmetic cvecs of Phase 1, we refer to them as *pvecs* (predicate vectors). Later, these pvecs will be used to find conditional candidates during *conditional cvec matching*.

In order for CHOMPY to use these pvecs to discover conditional candidates, CHOMPY creates P, a mapping from pvecs in E_c to their corresponding conditions. The GETPVECS procedure called on line 8 constructs P by grouping the e-classes in E_c by their pvecs and mapping each pvec to the canonical conditions of the equivalence classes in the group. A fragment of this mapping is shown in Figure 3, where, for example, the pvec $[T, F, F] \mapsto$ [`a < 0`], and $[T, T, F] \mapsto$ [`a <= 0`].

B. Implication Synthesis (Line 9)

Before generating conditional candidates, CHOMPY computes a set of implications I over the conditions in W_c.

CHOMPY uses I in two ways: (1) for tracking implications between conditions, and (2) to perform *implication propagation*, a technique which empowers CHOMPY to infer what assumptions follow from a base condition. Naively, given a condition workload W_c, a set of implications I could be constructed by using a validator (an SMT solver) to pairwise check if $c_1 \Rightarrow c_2$ for any two conditions $c_1, c_2 \in W_c$. However, constructing implications in this manner has $\mathcal{O}(|W_c|^2)$ complexity, and does not scale to large condition grammars.

Now, we describe how CHOMPY constructs I using an "implication graph" (E_I). A key insight of CHOMPY is that like rule synthesis, implication synthesis follows a pattern of enumeration, identification, and minimization. First, CHOMPY internally builds a new e-graph E_I from the conditions in E_c. To help scale candidate discovery, E_I is compressed using R, which unifies conditions like `a > 0` and `0 < a`. A slice of E_I can be found in the black portions of Figure 3 (Left). The next step, candidate finding, is similar to cvec matching. Candidate finding is done through *pvec matching* on the different e-classes in E_I. Specifically, two e-classes e_i, e_j with pvecs $[a_1, ..., a_n]$, $[b_1, ..., b_n]$ pvec match iff: $\forall i. a_i \Rightarrow b_i$.

Implication minimization. Conditions form a preorder under implication; CHOMPY maintains a graph representation of this preorder in the form of E_I, to track the information in I. As implications are added to I, CHOMPY reflects this in E_I through adding `edge`s and `path`s between conditions, where a `path` between two conditions means a series of `edge`s connects them. The right portion of Figure 3 shows two example graphs. The first one of these two represents CHOMPY in a state where $I = [\texttt{a <= -2} \Rightarrow \texttt{a < 0}, \texttt{a <= -2} \Rightarrow \texttt{a <= 0}]$. When these implications are "run", they create `edge`s between any conditions which match the left- and right-hand sides. In the figure, we represent these `edge`s as black arrows, such as the `edge` between `a <= -2` and `a < 0`. The rightmost graph in Figure 3 represents CHOMPY in a state where $I = [\texttt{a <= -2} \Rightarrow \texttt{a < 0}, \texttt{a < 0} \Rightarrow \texttt{a <= 0}]$. Minimization follows the same formula of popping candidates from the set one at a time. Once a candidate $p \Rightarrow q$ is popped, CHOMPY adds it to I if it is valid (checked via SMT). Then, CHOMPY runs I on E_I, i.e., it adds new `edge`s and `path`s to E_I. The remaining implication candidates are then reassessed: after the freshly popped implication is added to I, if a candidate $p' \Rightarrow q'$ has a `path` from p' to q', then the candidate is discarded.

The orange portions of Figure 3 (Right) show CHOMPY using existing implications to prune out redundant ones. In the graph on the left, CHOMPY is reassessing the usefulness of `a < 0` $\Rightarrow$ `a <= 0`. CHOMPY queries the graph for `path(a < 0, a <= 0)`. Because no path exists, CHOMPY keeps the implication as a candidate. In contrast, consider the state modeled by the other graph, where CHOMPY has popped `a <= -2` $\Rightarrow$ `a <= 0`. Like in the first example, CHOMPY queries this graph for `path(a <= -2, a <= 0)`. A series of edges linking `a <= -2` and `a <= 0` already exists, making CHOMPY discard the candidate.

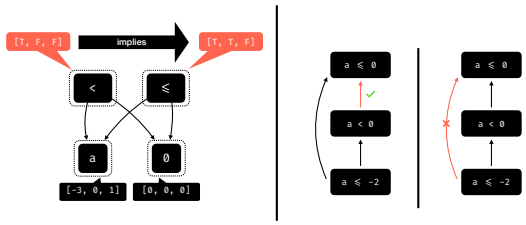

Fig. 3: Implication Synthesis. *Left:* in a condition e-graph, *pvec matching* identifies candidates for implications. *Right:* Implications are only accepted (red arrow with green check) into I when they express a newfound path between conditions. The graph on the left accepts the implication, and the one on the right rejects it.

Algorithm 2: High-level approach for minimization that CHOMPY generalizes for conditional rule minimization with predicate implications (orange). ADDEXPR returns the e-class id after adding a term.

Input: chosen ruleset R, candidates C, implications I, interpreter Φ
Output: minimized ruleset R'

```
1  R' ← R
2  while C ≠ ∅ do
3      r_new ← PopBest(C)
4      R' ← R' ∪ {r_new}
5      foreach rule = (if c then l ⇝ r) ∈ C do
6          E ← NewEGraph(Φ)
7          AddExpr(E, c)
8          E ← RunRules(E, R')
9          E ← ImplicationProp(E, I)
10         l' ← AddExpr(E, l)
11         r' ← AddExpr(E, r)
12         E ← RunRules(E, R')
13         if Find(E, l') == Find(E, r') then
14             C ← C \ {rule}
15         end
16     end
17 end
18 return R'
```

C. Conditional Candidate Identification (Line 10)

CHOMPY generates a set of conditional candidates C_c through conditional cvec matching. Two cvecs $[a_1, ..., a_n]$ and $[b_1, ..., b_n]$ *conditionally* match under pvec $[p_1, ..., p_n]$ if they are equal under the pvec. Formally, they match iff: $\forall i. p_i \Rightarrow (a_i = b_i)$. For each pair of e-classes (e_i, e_j) and for every predicate pattern $t_p \in P(pvec)$, CHOMPY generates a conditional candidate whenever the pvec witnesses a match between the two e-classes. The conditional candidate links each predicate pattern t_p with t_i and t_j, the canonical representatives of e_i and e_j. Before adding a candidate c to C_c, CHOMPY validates the candidate using an SMT solver (Z3). In the formula below, we write rule validation passing as $ok(t_p, t_i, t_j)$.

$$C_c = \Big\{ \texttt{if } t_p \texttt{ then } t_i \rightsquigarrow t_j \;\Big|\; e_i,\, e_j \in eclasses(E) \wedge e_i \neq e_j;\; pvec \in dom(P),\; t_p \in P(pvec),\; \text{match}(e_i, e_j, pvec),\; ok(t_p, t_i, t_j) \Big\}$$

Adding conditional logic often produces multiplicatively

more candidate rules, even for a small grammar. Even in our simple example, where Phase 1 produced only a handful of total rules, Phase 2 generates a larger set of conditional candidates, including:

```
if a < b then min(a, b) ⇝ a,
if b < a then min(a, b) ⇝ b,
if a <= b then min(a, b) ⇝ a,
if b <= a then min(a, b) ⇝ b,
if a <= 0 then min(a + b, b) ⇝ a + b,
if a <= -2 then min(a + b, b) ⇝ a + b
```

D. Conditional Rule Minimization (Line 11)

The final step in Phase 2 is to reduce the set of conditional candidates C_c to a compact, non-redundant subset using I computed in the previous step. Algorithm 2 describes CHOMPY's minimization algorithm for conditional rules. Rules from C_c are popped one at a time according to a cost function [2], and added to R' (see its definition in Algorithm 2). As rules are added, CHOMPY assesses the remaining conditional rules in C_c for redundancy, discarding those that are *derivable* from running the rules in R'.

Since R' contains conditional rules, we briefly explain CHOMPY's representation of them. E-graphs are typically used for applying unconditional rewrite rules: to support conditional rewriting, CHOMPY adds explicit *assumption nodes* [12] to the e-graph that "unlock" the rewriting power of conditional rules. For example, introducing an assumption node `assume(a != 0)` allows the conditional rule `if x != 0 then x / x ⇝ 1` to merge `a / a` and `1`. Assessing conditional rules for redundancy introduces added nuance. To illustrate this nuance, we show how traditional *derivability* techniques [1], [2] do not properly address conditional rules. Consider the following two rules from C_c:

```
if a <= b then min(a, b) ⇝ a,
if a < b then min(a, b) ⇝ a.
```

Suppose we have already added the first rule to R'. Should we keep the second? *No*, because the second rule's condition implies that of the first. The second rule is *stricter* and the first is more general; the first rule can rewrite anything that the second rule matches on. Traditional rule minimization used in prior work [1], [2] lacks such reasoning about condition implication. Recall that prior work assesses a rule's redundancy by adding the rule's left- and right-hand sides to a fresh e-graph (here, `min(a, b)`, `a`). Even if the e-graph is seeded with the assumption `a < b`, the first rule, which only looks for `a <= b`, will not be "unlocked". This unlocking of conditional rules is precisely why CHOMPY needs I.

Conditional Derivability. The above example shows that standard rule application is not sufficient for *deriving* conditional rules and minimizing conditional rulesets; it cannot derive rules whose side conditions differ by logical implication. Conditional rule minimization requires using *predicate implications* that inform the e-graph that more general assumptions follow from specific ones. This observation motivates our notion of *conditional derivability*. Conditional derivability concerns a ruleset A, a candidate rule $cand :=$ `if` c `then` $l \rightsquigarrow r$, and a set of implications I. If I suggests that c implies some other condition c', and under c', A can merge l and r, then A derives $cand$ ($A \models cand$) and CHOMPY can discard $cand$. More formally, $A \models$ `if` c `then` $l \rightsquigarrow r$ if there exists c' such that $I \vdash c \Rightarrow c'$ and A merges l and r in an e-graph seeded with `assume(c')`. In CHOMPY, we use the implications in the last step to fire Datalog-style rules that add `assume` nodes to the e-graph. We refer to this as implication propagation. For example, the implication `a < b` $\Rightarrow$ `a <= b` manifests as a rule which searches for any assumption matching `assume(a < b)` and adds `assume(a <= b)` to the e-graph.

CHOMPY uses *implication propagation* to perform conditional rule minimization, as shown in Algorithm 2. This algorithm builds on top of prior work, with the exceptions of Lines 7–9, which (1) add a candidate's predicate to the e-graph, and (2) populate an assumption space with all assumptions that follow from that predicate.

Going back to our example, say that CHOMPY has discovered several predicate implications, including: `a < b` $\Rightarrow$ `a <= b`. With the rule `if a <= b then min(a, b)` $\rightsquigarrow$ `a` added to the ruleset, CHOMPY assesses the redundancy of `if a < b then min(a, b)` $\rightsquigarrow$ `a`. CHOMPY adds `a` and `min(a, b)` to an e-graph. Running Line 7 of Algorithm 2 seeds the e-graph with `assume(a < b)`. Line 8 adds alternate representations of the assumption to the e-graph, such as `assume(b > a)`, allowing CHOMPY to apply implications across them. Line 9 runs *implication propagation*, which matches on `assume(a < b)` and adds `assume(a <= b)` to the e-graph. This additional assumption allows the general rule to fire, merging the left- and right-hand sides, and allowing CHOMPY to discard the conditional rule. This process is repeated as new rules are added. With the implications, 4 out of 6 example candidates are discarded, leaving only the 2 which subsume the rest: `if a <= b then min(a, b)` $\rightsquigarrow$ `a` and `if a <= 0 then min(a + b, b)` $\rightsquigarrow$ `a + b`.

IV. Implementation and Evaluation

CHOMPY is implemented on top of ENUMO [1] in approximately 1600 lines of Rust. Following prior work [1], [2], we evaluate CHOMPY on a Halide-inspired grammar over operators:
`[+, -, *, /, %, min, max, <, <=, >, >=, ==, !=, &&]`,
using 13 term and 3 condition workloads (Table II in Appendix A).

All experiments were run on a machine with 32 GB RAM running Windows 11. Our targets are conditional rules from Caviar [6] (45 rules) and the Halide compiler [7] (84 rules), filtered to operators supported by CHOMPY and validated by

[2]This heuristic is a lexicographical ordering preferring rules with (1) more variables, (2) smaller AST cost, (3) weak preconditions (count of `T` in the condition's pvec.) Total rules have a `pvec_false` count of 0.

TABLE I: Conditional derivability scores across configurations of CHOMPY. Values are mean ± standard deviation over 5 runs. Column 2 shows all rules, column 3 shows only conditional rules. The numbers in parentheses indicate the sizes of the handwritten Caviar and Halide rulesets, respectively.

Configuration	Ruleset Size	Cond Size	% Caviar (45)	% Halide (84)	Runtime (s)
`baseline`	1579.0 ± 0.0	1174.0 ± 0.0	73.3 ± 0.0	57.1 ± 0.0	1551.7 ± 13.9
`llm_only`	116.4 ± 11.5	40.0 ± 14.5	9.8 ± 5.8	3.1 ± 2.5	30.7 ± 3.4
`llm_terms_conds`	207.6 ± 18.5	117.0 ± 15.6	25.8 ± 10.3	14.8 ± 12.6	551.9 ± 41.5
`llm_with_enum`	1526.4 ± 20.0	1118.8 ± 14.1	73.3 ± 0.0	58.3 ± 1.2	1676.0 ± 47.3
`llm_filter_1`	882.6 ± 22.3	477.6 ± 22.3	66.2 ± 3.7	55.5 ± 2.6	2176.5 ± 57.5
`llm_filter_5`	1430.0 ± 15.8	1025.0 ± 15.8	71.1 ± 2.2	59.8 ± 2.4	2269.0 ± 140.3
`llm_terms_filter`	1403.4 ± 61.7	992.4 ± 59.0	73.3 ± 1.6	60.0 ± 1.4	2333.8 ± 139.5

Z3 (with 1 second timeout) [3]. CHOMPY is open-source and is available at `https://github.com/ninehusky/chompy`. All rules produced by CHOMPY are verified by Z3 as mentioned in Section III.

We answer three research questions:

- **Q1** Can CHOMPY recover handwritten conditional rules from real systems (Caviar and Halide)?
- **Q2** How does CHOMPY compare to LLM-based approaches?
- **Q3** How important is implication propagation to making CHOMPY work?

A. Q1: Recovering handwritten conditional rules

Table I reports on derivability, i.e., the fraction of expert rules derivable from the synthesized ruleset across all configurations, averaged over 5 runs. CHOMPY's `baseline` recovers 73.3% of Caviar's and 57.1% of Halide's conditional rules on every run. This shows that CHOMPY is effective for synthesizing conditional rules that can derive a large portion of expert-written conditional rules from two real-world systems. We measure rule derivability, not downstream impact; the latter is left to future work. Section V-B discusses ruleset quality qualitatively.

B. Q2: CHOMPY vs. LLM-based approaches

A natural question, given the progress of LLMs, is whether prompting an LLM to "generate rewrite rules for this grammar" already solves the problem. We investigate this directly. All our LLM components use GPT-5.4 and the full prompts are in Appendix A.

First, we compare against an `llm_only` baseline, prompted to "generate rewrite rules for equality saturation" over the same grammar; each output rule is validated by Z3. `llm_only` is faster since it requires a single API call but it produces only 116.4 ± 11.5 sound rules, recovering 9.8% of Caviar's and just 3.1% of Halide's rules. CHOMPY beats this baseline by 63.5 and 54.0 percentage points respectively, with deterministic output (Table I).

Next, we evaluate CHOMPY augmented with LLM components in several ways. Replacing CHOMPY's entire enumeration with only LLM-suggested terms and conditions (`llm_terms_conds`) hinders rule discovery ($25.8\%/14.8\%$, with high variance across runs). Augmenting CHOMPY's enumeration with additional LLM-suggested terms (`llm_with_enum`) ties `baseline` on Caviar (73.3%) and slightly improves Halide (58.3%) at marginally smaller ruleset size.

We also used LLMs to reduce ruleset size in addition to derivability. When a minimized ruleset's size $|R|$ exceeds a threshold, CHOMPY prompts an LLM to partition R into semantic categories (e.g., "simplifications of `+` and `min`") and retains the top k rules per category. `llm_filter_5` reduces ruleset size by 9.4% while improving Halide derivability to 59.8%; aggressive filtering (`llm_filter_1`) cuts the ruleset nearly in half (883 rules) but sacrifices derivability.

Combining LLM-augmented enumeration with filtering (i.e., `llm_terms_filter`), which uses `llm_with_enum` with `llm_filter_5` is the best overall configuration. It matches `baseline` on Caviar (73.3%) and achieves the highest Halide derivability (60.0%) at an 11% smaller ruleset. This suggests LLM-based semantic categorization prunes low-value conditional rules with minimal loss in coverage.

Note that as shown in Table I, the LLM results have considerable variance across runs which makes it difficult to make stronger claims about their usefulness. Overall, our main takeaway is that LLMs alone are not a substitute for principled rule synthesis[4], but they are effective when combined with synthesis in a structured way. We found that semantic rule categorization for minimization yields small, high-derivability rulesets. The variance across runs also made it difficult to debug and reproduce the results from the LLM-based treatments.

Timing and Validation: We report on the time spent by CHOMPY in various parts of the algorithm and the number of solver timeouts CHOMPY encountered. In CHOMPY's baseline configuration, Phase 1 (total rule synthesis) accounts for 3.1% of the overall runtime, while Phase 2 (conditional rule synthesis) accounts for the remaining 96.9%. This difference is expected: the conditional equality space is substantially larger than the total equality space. Minimization collectively accounts for 15.0% of the overall runtime.

During baseline synthesis, CHOMPY issued 90,322 SMT queries, of which only 14 (0.016%) timed out.

C. Q3: Sensitivity Study — Effect of Implication Propagation

A key component of CHOMPY is implication propagation. To assess the importance of implications in CHOMPY's flow, we compare the full system against a version with this component removed (Line 9 in Algorithm 2). With implication propagation, CHOMPY terminates in 1551.7 seconds and produces 1579 rules (see results over 5 runs in Table I). Without implication propagation, CHOMPY did not terminate even after running for several hours; in its last logged state it had generated 3931 rules, at which point we stopped the run. This suggests that without implication propagation, rule exploration of useful grammars is impractical.

[3]We encoded Halide's division semantics to the best of our ability when using Z3 to validate the rules. However, several Halide rules did not validate within the set timeout and were discarded.

[4]Over the duration of this project, we spent $116.62 in API credits for using OpenAI's models.

V. Discussion

In this section we discuss various optimizations our implementation includes that we elided in Section III and reflect on ruleset quality.

A. Optimization.

There are a few optimizations in our implementation of CHOMPY that, for readability, we elided from Section III. First, the conditions in W_c often contain syntactic variations of the same predicate, e.g., `a < 0` and `0 > a`. In practice, our implementation applies the same compression step to E_c as is described for E_I in Section III-B. The candidates removed in this fashion are those which would be removed by minimization at a later phase, but eliminating them before generation reduces the need for unnecessary usage of the validator.

The second optimization involves threading implications into conditional cvec matching before invoking the validator. In practice, equality assumptions are especially effective here: our implementation takes `assume(a == b)` and merges `a` and `b`, collapsing both sides of many candidate pairs. A promising extension is to use the synthesized implications more aggressively at this stage: candidates whose conditions are stronger than those of already-accepted rules could be pruned during generation rather than during minimization to reduce validator load.

B. Ruleset Quality

Derivability tells us what equalities are recoverable by CHOMPY, but not whether the ruleset is *good*. A ruleset can score well on derivability metrics while containing rules that do not fire, or unnecessarily expand the e-graph. Measuring downstream performance requires plugging CHOMPY's rules into a rewrite system, which we leave for future work. In this section, we instead characterize CHOMPY's ruleset qualitatively by what rules CHOMPY's ruleset recovers, and what rules CHOMPY misses. As shown in Section IV-A, CHOMPY recovers many conditional equalities in handwritten rulesets. The rules that CHOMPY can recapture include many "textbook-style" rules which are easy to express using handwritten workloads. These handwritten workloads also allow CHOMPY to synthesize many rules absent from handwritten rules: for example,

```
if (c <= 0) then min((c * b), (c * a)) ⇝
                                 c * (max(b, a))
```

allows for sign-aware distribution of a multiplier over `min` / `max`. The rules that CHOMPY fails to synthesize in our evaluation are those less amenable to bottom-up enumeration. Such rules often capture bespoke equalities which rely on using unusual term patterns or side conditions that are difficult to express in a general-purpose workload or cost function. For example, CHOMPY does not synthesize the following rule from Caviar:

```
if (b > 0) then ((a * b) < c) ⇝
        (a < (((c + b) - 1) / b))
```

Although one could specialize the workload or cost function to discover these individual rules, doing so may reduce the generality of the synthesizer or cause it to miss other classes of rules.

C. Limitations

CHOMPY is limited by the search space of its workloads, particularly for conditional rules whose profitable side conditions are often larger than the terms they rewrite (e.g., Alive-Infer [13], which synthesizes such conditions per-rule). Our LLM use is restricted to prompt engineering; specialized training can potentially improve results. Using CHOMPY for larger domains is left for future work.

Future work will evaluate synthesized rulesets by downstream performance in a rewrite system, rather than by derivability alone.

VI. Related Work

CHOMPY directly extends rewrite rule synthesis tools that use equality saturation. RULER pioneered the use of e-graphs for discovering rules [2], while ENUMO builds on RULER's framework to allow developers to express sketches of the program space over which rules are synthesized. Theory exploration itself is a well-studied topic [4], [14], [15]. In particular, THESY [4] uses e-graphs for symbolic observational equivalence to reduce the space of equalities, similar to RULER. The main focus of these tools, however, has been on total rewrite rules. SWAPPER [16] requires a corpus of benchmark formulas from a target domain, samples frequent patterns, and synthesizes one rule per pattern. CHOMPY requires only a grammar and minimizes rulesets using implication propagation. SWAPPER's implication reasoning is internal to a single rule's predicate search; cross-rule condition implication is unique to CHOMPY. ALIVE-INFER synthesizes the weakest side condition under which a given rewrite is sound [13]. Similarly, ROVER [17] generates necessary and sufficient conditions under which RTL rewrites are valid. Other work combines conditions and e-graphs for case-splitting. CCLEMMA [3] implements case-splitting by unioning variables with different variants of inductive datatypes, and EASTER EGG [5] efficiently represents many separate conditions in a single e-graph data structure. Prior work [11], [18] has also presented tools for automatic peephole optimization synthesis and used fingerprints (similar to cvecs) to detect potential candidates ([11]).

Acknowledgment

We thank the reviewers for their helpful and thorough feedback. We are grateful to Hila Peleg for giving us advice on our evaluation plan, to Cole Kurashige and Shaan Nagy for numerous brainstorming sessions, and to Anjali Pal for helping debug Enumo.

DATA AVAILABILITY

The artifact(s) associated with this paper is/are available at Zenodo under the following DOI: 10.5281/zenodo.21727983.

REFERENCES

[1] A. Pal, B. Saiki, R. Tjoa, C. Richey, A. Zhu, O. Flatt, M. Willsey, Z. Tatlock, and C. Nandi, "Equality saturation theory exploration à la carte," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA2, Oct. 2023. [Online]. Available: https://doi.org/10.1145/3622834

[2] C. Nandi, M. Willsey, A. Zhu, Y. R. Wang, B. Saiki, A. Anderson, A. Schulz, D. Grossman, and Z. Tatlock, "Rewrite rule inference using equality saturation," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, Oct. 2021. [Online]. Available: https://doi.org/10.1145/3485496

[3] C. Kurashige, R. Ji, A. Giridharan, M. Barbone, D. Noor, S. Itzhaky, R. Jhala, and N. Polikarpova, "Cclemma: E-graph guided lemma discovery for inductive equational proofs," *Proc. ACM Program. Lang.*, vol. 8, no. ICFP, Aug. 2024. [Online]. Available: https://doi.org/10.1145/3674653

[4] E. Singher and S. Itzhaky, "Theory exploration powered by deductive synthesis," in *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*. Berlin, Heidelberg: Springer-Verlag, 2021, pp. 125–148. [Online]. Available: https://doi.org/10.1007/978-3-030-81688-9_6

[5] ——, "Easter egg: Equality reasoning based on e-graphs with multiple assumptions," in *2024 Formal Methods in Computer-Aided Design (FMCAD)*, 2024, pp. 70–83.

[6] S. Kourta, A. Namani, F. B. Tayeb, K. M. Hazelwood, C. Cummins, H. Leather, and R. Baghdadi, "Caviar: An e-graph based TRS for automatic code optimization," *CoRR*, vol. abs/2111.12116, 2021. [Online]. Available: https://arxiv.org/abs/2111.12116

[7] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *SIGPLAN Not.*, vol. 48, no. 6, Jun. 2013. [Online]. Available: https://doi.org/10.1145/2499370.2462176

[8] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: a theorem prover for program checking," *J. ACM*, vol. 52, no. 3, May 2005. [Online]. Available: https://doi.org/10.1145/1066100.1066102

[9] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "egg: Fast and extensible equality saturation," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, Jan. 2021. [Online]. Available: https://doi.org/10.1145/3434304

[10] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: a new approach to optimization," in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '09. New York, NY, USA: Association for Computing Machinery, 2009. [Online]. Available: https://doi.org/10.1145/1480881.1480915

[11] S. Bansal and A. Aiken, "Automatic generation of peephole superoptimizers," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 5, Oct. 2006. [Online]. Available: https://doi.org/10.1145/1168917.1168906

[12] S. Coward, G. A. Constantinides, and T. Drane, "Automating constraint-aware datapath optimization using e-graphs," 2023. [Online]. Available: https://arxiv.org/abs/2303.01839

[13] D. Menendez and S. Nagarakatte, "Precondition inference for peephole optimizations in llvm," 2017. [Online]. Available: https://arxiv.org/abs/1611.05980

[14] M. Johansson, D. Rosén, N. Smallbone, and K. Claessen, "Hipster: Integrating theory exploration in a proof assistant," in *Intelligent Computer Mathematics*, S. M. Watt, J. H. Davenport, A. P. Sexton, P. Sojka, and J. Urban, Eds. Cham: Springer International Publishing, 2014, pp. 108–122.

[15] K. Claessen, N. Smallbone, and J. Hughes, "Quickspec: Guessing formal specifications using testing," in *Tests and Proofs*, G. Fraser and A. Gargantini, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 6–21.

[16] R. Singh and A. Solar-Lezama, "Swapper: a framework for automatic generation of formula simplifiers based on conditional rewrite rules," in *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '16. Austin, Texas: FMCAD Inc, 2016.

[17] S. Coward, T. Drane, and G. A. Constantinides, "Rover: Rtl optimization via verified e-graph rewriting," 2024. [Online]. Available: https://arxiv.org/abs/2406.12421

[18] J. W. Davidson and C. W. Fraser, "Automatic generation of peephole optimizations," *SIGPLAN Not.*, vol. 39, no. 4, pp. 104–111, Apr. 2004. [Online]. Available: https://doi.org/10.1145/989393.989407

APPENDIX

The grammar we used in our evaluation is shown in Table II. Note that the grammar is sorted: arithmetic expressions and Boolean predicates are generated separately. Boolean operators only compose predicates, while arithmetic operators only compose integer expressions. The burden of ensuring this is on the user writing the ENUMO program that generates the workloads.

TABLE II: Overview of term workloads, their operators, and total rewrites. Most term workloads are paired with the same conditional workload: comparisons of `(<, <=, ==, !=)` over `{a, b, c, 0}`. There are three exceptions: `lt_add_min` and `lt_add_max` use conditions built from `<`, the variables `a-d`, and 0. `mul_div_mod` is run with conditions containing `<` and the divisibility conditions `(&& (< 0 x) (== (% a b) 0))` for `x ∈ {a, b}`.

Name	Operators	Rules Learned
base_comps	<, <=, ==, !=	20
mul_div_mod	*, /, %	37
simp_comps	+, -, <, >	82
arith_basic	+, -, *, /	84
and_comps	&&, comps	126
min_max	min, max	13
min_max_add	min, max, +	6
eq_simp_min	min, ==	7
eq_simp_max	max, ==	6
min_max_mul	min, max, *	53
min_max_div	min, max, /	410
lt_add_min	<, +, min	341
lt_add_max	<, +, max	394

For reproducibility, we include the prompts used in our LLM-assisted experiments.

A. Term Enumeration Prompt

```
You are generating new terms in a small programming language.
The syntax is s-expressions with the following operators:
{operators}

You will be given a list of existing terms.
Your task is to propose additional terms that are:
- significantly different from the input set,
- likely to be useful in rewrite rules (e.g., distributivity, commutativity, associativity, idempotence),
- non-trivial (avoid constant-only terms unless they reduce to a single constant),
- and include variables in most cases.

Output requirements:
- Generate about {term_limit} additional terms.
- Produce a variety of sizes and shapes (not just shallow terms, include large terms as well).
- Ensure all terms are distinct and not repeats of the input.
- Output **only the terms**, one per line.
- Do not include any commentary, numbering, or explanations.
- Do not introduce any new variables outside the set given.
- Do not include any markdown backticks.
- Do not use the token `assume` anywhere in your output -- it is a reserved internal keyword and any term containing it will be discarded.

Example Input Terms:
x
y
0
1
(+ x y)
(abs x)

Example Output:
(+ (* x y) (* x z))
(* (+ x y) z)
(max (abs y) x)
(min x (+ y 1))
(- (* x 0) y)

Input Terms:
{input_text}
```

B. Condition Enumeration Prompt

```
You are generating logical conditions in a small programming language.
The syntax is s-expressions with the following operators:
{operators}

You will be given a list of terms (expressions).
Your task is to propose additional conditions that are:
- syntactically valid Boolean expressions,
- useful for rewrite rules (e.g., inequalities, equalities, divisibility checks, non-negativity tests),
- non-trivial (avoid tautologies or contradictions like `(< x x)`),
- structurally diverse (use different operators and nesting where possible),
- and involve variables where possible.

Conditions can be:
- simple comparisons (like `(< x y)` or `(== a 0)`), or
- compound conditions using logical operators such as `(&& cond1 cond2)`
```

```
    or `(|| cond1 cond2)` when this makes
     them more useful for rewrite rules.
- assume that variables are numbers, don'
    t enumerate stuff like `(&& a b)`.

Output requirements:
- Generate about {cond_limit} distinct
    conditions.
- Each condition must evaluate to a
    Boolean (true/false).
- Conditions should compare or test terms
     using operators such as <, <=, >,
    >=, ==, !=, and also include useful
    checks (e.g., remainder equal to zero
    , comparisons to constants).
- Output **only the conditions**, one per
     line.
- Do not repeat the input terms.
- Do not include commentary, numbering,
    or explanations.
- Do not include any markdown backticks.
- Do not use the token `assume` anywhere
    in your output -- it is a reserved
    internal keyword and any condition
    containing it will be discarded.

Example Input Terms:
a
b
(* a b)
(% a b)

Example Output Conditions:
(< a 0)
(>= (* a b) 1)
(== (% a b) 0)
(!= a b)
(<= b (* a b))
(&& (> a 0) (< b a))

Input Terms:
{input_text}
```

C. Rule Generation Prompt

The LLM was prompted with a superset grammar; generated rules using unsupported operators were discarded before validation.

```
You are an expert in program optimization
     and algebraic reasoning.

Generate a diverse set of critical
    conditional rewrite rules
for an equality saturation rewrite system
    .

Operators (all integer-valued unless
    noted):
- (+ a b) addition
- (- a b) subtraction
- (* a b) multiplication
- (/ a b) integer division (truncates
    toward zero)
- (% a b) remainder
- (min a b) minimum
- (max a b) maximum
- (abs a) absolute value
- (|| a b) boolean OR
- (&& a b) boolean AND
- (! a) boolean NOT

Boolean-valued operators (conditions are
    limited to just these; expressions
    can contain them):
- (< a b) strictly less than
- (<= a b) less than or equal
- (== a b) equal
- (!= a b) not equal
- (> a b) strictly greater than
- (>= a b) greater than or equal
- (&& a b) logical and
- (|| a b) logical or
- (! a) logical not

Variables: ?a ?b ?c ?d
Constants: any integer literal (e.g. 0,
    1, -1, 2)

Rule format - exactly one of:
  LHS ==> RHS
  LHS ==> RHS if COND

All terms are s-expressions. Variables
    must be prefixed with `?`.
Do NOT output bidirectional rules (<=>),
    explanations, or any other text.
Do NOT use the token `assume` anywhere in
     your output - it is a reserved
    internal keyword and any rule
    containing it will be discarded.

Example rules:
(+ ?a 0) ==> ?a
(* ?a 1) ==> ?a
(- ?a ?a) ==> 0
(/ ?a ?a) ==> 1 if (!= ?a 0)
(min ?a ?b) ==> ?a if (<= ?a ?b)
(max ?a ?b) ==> ?b if (<= ?a ?b)
(* ?a (+ ?b ?c)) ==> (+ (* ?a ?b) (* ?a ?
    c))
(% ?a ?b) ==> 0 if (== (% ?a ?b) 0)
(abs ?a) ==> ?a if (>= ?a 0)
(abs ?a) ==> (* -1 ?a) if (< ?a 0)

It's okay for the conditions to be large,
as long as they are valid and potentially
     useful for optimization.
Output only the rules, one per line.

See the attached appendix for a concrete
    interpreter
which encodes the semantics of the
    operators above.

========== APPENDIX ============
```

```
fn eval<'a, F>(&'a self, cvec_len: usize, mut get_cvec: F) -> CVec<Self>
where
    F: FnMut(&'a Id) -> &'a CVec<Self>,
{
    let one = 1.to_i64().unwrap();
    let zero = 0.to_i64().unwrap();
    match self {
        Pred::Lit(c) => vec![Some(*c); cvec_len],
        Pred::Abs(x) => map!(get_cvec, x => Some(x.abs())),
        Pred::Lt([x, y]) => {
            map!(get_cvec, x, y => if x < y {Some(one)} else {Some(zero)})
        }
        Pred::Leq([x, y]) => {
            map!(get_cvec, x, y => if x <= y {Some(one)} else {Some(zero)})
        }
        Pred::Gt([x, y]) => {
            map!(get_cvec, x, y => if x > y {Some(one)} else {Some(zero)})
        }
        Pred::Geq([x, y]) => {
            map!(get_cvec, x, y => if x >= y {Some(one)} else {Some(zero)})
        }
        Pred::Eq([x, y]) => {
            map!(get_cvec, x, y => if x == y {Some(one)} else {Some(zero)})
        }
        Pred::Neq([x, y]) => {
            map!(get_cvec, x, y => if x != y {Some(one)} else {Some(zero)})
        }
        Pred::Implies([x, y]) => {
            map!(get_cvec, x, y => {
              let xbool = *x != zero;
              let ybool = *y != zero;
              if !xbool || ybool {Some(one)} else {Some(zero)}
            })
        }
        Pred::Not(x) => {
            map!(get_cvec, x => if *x == zero { Some(one)} else {Some(zero)})
        }
        Pred::Neg(x) => map!(get_cvec, x => Some(-x)),
        Pred::And([x, y]) => {
            map!(get_cvec, x, y => {
               let xbool = *x != zero;
               let ybool = *y != zero;
               if xbool && ybool { Some(one) } else { Some(zero) }
            })
        }
        Pred::Or([x, y]) => {
            map!(get_cvec, x, y => {
               let xbool = *x != zero;
               let ybool = *y != zero;
               if xbool || ybool { Some(one) } else { Some(zero) }
            })
        }
        Pred::Xor([x, y]) => {
            map!(get_cvec, x, y => {
               let xbool = *x != zero;
               let ybool = *y != zero;
               if xbool ^ ybool { Some(one) } else { Some(zero) }
            })
        }
        Pred::Add([x, y]) => map!(get_cvec, x, y => x.checked_add(*y)),
        Pred::Sub([x, y]) => map!(get_cvec, x, y => x.checked_sub(*y)),
        Pred::Mul([x, y]) => map!(get_cvec, x, y => x.checked_mul(*y)),
        Pred::Div([x, y]) => map!(get_cvec, x, y => {
           if *y == 0 {
              Some(0)
           } else if *x == i64::MIN && *y == -1 {
              // Wraparound case
              Some(i64::MIN)
           } else {
              Some(x.div_euclid(*y))
           }
        }),
        Pred::Mod([x, y]) => map!(get_cvec, x, y => {
           if *y == zero {
              Some(zero)
           } else {
              Some(x.rem_euclid(*y))
           }
        }),
        Pred::Min([x, y]) => map!(get_cvec, x, y => Some(*x.min(y))),
        Pred::Max([x, y]) => map!(get_cvec, x, y => Some(*x.max(y))),
        Pred::Select([x, y, z]) => map!(get_cvec, x, y, z => {
          let xbool = *x != zero;
          if xbool {Some(*y)} else {Some(*z)}
        }),
    }
}
```

D. Semantic Rule Categorization Prompt

```
You are the world's leading expert in program optimization and algebraic reasoning.
You are helping organize rewrite rules for use in an equality saturation system.

You will be given a batch of valid candidate rewrite rules.

Your goal:
 - Group the rules into semantic categories according to what they do.
 - If a rule produces a RHS that is **structurally smaller than the LHS** (fewer operators, less nesting), place it in its own category.
 - If a rule produces a RHS that is **more canonical or normalized** (terms reordered into standard order, consistent structure), place it in its own category.
 - If a rule produces a RHS that is **more balanced or symmetric** than the LHS, place it in its own category.
 - Do not lump simplifications with rules that merely reshuffle or perform minor transformations.
 - Single out important rewrite patterns like idempotency, distributivity, or constant simplifications: they should always get their own category, even if there are only one or two rules of that type.
 - Do not group rules together just because they use the same operators; create new categories for meaningful semantic differences.
 - Place each rule under exactly one category.
 - Remove rules which just look like noise.
 - Strive to capture semantic differences clearly, even if that results in more categories than before.
 - Avoid redundant categories; rules that are semantically similar should be grouped together.

Format your output like this:

Category: <brief description of transformation>
<rule 1>
<rule 2>
...

Category: <next transformation>
<rule 1>
<rule 2>
...

Do not output numeric hints, explanations, or any other text. Only output the categories and rules.

Example Input:
(+ (* ?a 1) ?b) ==> (+ ?b (* ?a 1))
(/ ?a ?a) ==> 1 if (!= ?a 0)
(+ ?a (+ ?b ?c)) ==> (+ ?a (+ ?c ?b))
(min ?a (+ ?a ?b)) ==> (+ ?a ?b) if (< ?b 0)
(+ (+ ?a ?b) (* ?c 1)) ==> (+ (* ?c 1) (+ ?b ?a))
(max (+ ?a ?b) ?c) ==> (max (+ ?b ?a) ?c)
(< ?a (+ ?a ?b)) ==> (< ?a (+ ?b ?a))
(* ?a (* ?b ?c)) ==> (* ?a (* ?c ?b))
(+ (+ ?a ?b) (+ ?c ?d)) ==> (+ (+ ?b ?a) (+ ?d ?c))
(< (?a (+ ?b ?c))) ==> (< ?a (+ ?c ?b)) if (!= ?b 0)
(min ?a ?b) ==> ?a if (< ?a ?b)
(* (* ?a ?b) ?c) ==> (* (* ?b ?a) ?c)
(< (+ ?a ?b) (+ ?b ?a)) ==> 0 if (== ?a ?b)
(min (+ ?a ?b) ?c) ==> (+ ?a (min ?b ?c)) if (< ?a ?c)
(+ ?a (* ?b 0)) ==> ?a if (>= ?b 0)
(+ ?a (+ ?b ?c)) ==> (+ ?b (+ ?a ?c))
(< (+ ?a ?b) (+ ?b ?a)) ==> (< (+ ?b ?a) (+ ?a ?b))
(/ ?a ?a) ==> 1 if (> ?a 0)
(min ?a (+ ?b ?c)) ==> (min (+ ?c ?b) ?a)
(* ?a (+ ?b ?c)) ==> (+ (* ?a ?b) (* ?a ?c)) if (>= ?a 0)
(+ (+ ?a ?b) ?c) ==> (+ (+ ?b ?a) ?c)
(min ?a (+ ?b ?c)) ==> (min ?a (+ ?c ?b))

Example Output (Good):
Category: Simplifications for Division
(/ ?a ?a) ==> 1 if (!= ?a 0)
(/ ?a ?a) ==> 1 if (> ?a 0)

Category: Simplifications for Min
(min ?a ?b) ==> ?a if (< ?a ?b)
(min ?a (+ ?a ?b)) ==> (+ ?a ?b) if (< ?b 0)

Category: Constant Multiplication and Addition Simplifications
(+ ?a (* ?b 0)) ==> ?a if (>= ?b 0)
(+ (* ?a 1) ?b) ==> (+ ?b (* ?a 1))
(+ (+ ?a ?b) (* ?c 1)) ==> (+ (* ?c 1) (+ ?b ?a))

Category: Commutativity and Argument Shuffling
(+ ?a (+ ?b ?c)) ==> (+ ?a (+ ?c ?b))
(+ ?a (+ ?b ?c)) ==> (+ ?b (+ ?a ?c))
(* ?a (* ?b ?c)) ==> (* ?a (* ?c ?b))
(* (* ?a ?b) ?c) ==> (* (* ?b ?a) ?c)
(+ (+ ?a ?b) (+ ?c ?d)) ==> (+ (+ ?b ?a) (+ ?d ?c))
(+ (+ ?a ?b) ?c) ==> (+ (+ ?b ?a) ?c)
(max (+ ?a ?b) ?c) ==> (max (+ ?b ?a) ?c)
```

```
79
80 Category: Inequality Shuffles
81 (< ?a (+ ?a ?b)) ==> (< ?a (+ ?b ?a))
82 (< (+ ?a ?b) (+ ?b ?a)) ==> (< (+ ?b ?a)
       (+ ?a ?b))
83 (< (+ ?a ?b) (+ ?b ?a)) ==> 0 if (== ?a ?
       b)
84 (< (?a (+ ?b ?c))) ==> (< ?a (+ ?c ?b))
       if (!= ?b 0)
85
86 Category: Min/Max Transformations
87 (min (+ ?a ?b) ?c) ==> (+ ?a (min ?b ?c))
        if (< ?a ?c)
88 (min ?a (+ ?b ?c)) ==> (min (+ ?c ?b) ?a)
89 (min ?a (+ ?b ?c)) ==> (min ?a (+ ?c ?b))
90
91 Category: Distributivity of
       Multiplication over Addition
92 (* ?a (+ ?b ?c)) ==> (+ (* ?a ?b) (* ?a ?
       c)) if (>= ?a 0)
93
94 Category: Canonicalization / Reordering
95 (+ ?a (+ ?b ?c)) ==> (+ ?a (+ ?c ?b))
96 (+ ?a (+ ?b ?c)) ==> (+ ?b (+ ?a ?c))
97 (max (+ ?a ?b) ?c) ==> (max (+ ?b ?a) ?c)
98
99 Category: Structural Simplifications
100 (min ?a (+ ?a ?b)) ==> (+ ?a ?b) if (< ?b
        0)
101 (/ ?a ?a) ==> 1 if (!= ?a 0)
102 (+ ?a (* ?b 0)) ==> ?a if (>= ?b 0)
103
104 Category: Balanced / Symmetric
       Transformations
105 (+ (+ ?a ?b) (+ ?c ?d)) ==> (+ (+ ?b ?a)
       (+ ?d ?c))
106 (+ (+ ?a ?b) ?c) ==> (+ (+ ?b ?a) ?c)
107
108 Input:
109 candidates_text
```