Optimizing Regular Expressions via Rewrite-Guided Synthesis

Jedidiah McClurg
Colorado State University
USA

Miles Claver[*]
Colorado School of Mines
USA

Jackson Garner[*]
Colorado School of Mines
USA

Jake Vossen[*]
Colorado School of Mines
USA

Jordan Schmerge
Colorado School of Mines
USA

Mehmet E. Belviranli
Colorado School of Mines
USA

ABSTRACT

Regular expressions are pervasive in modern systems. Many real-world regular expressions are inefficient, sometimes to the extent that they are vulnerable to complexity-based attacks, and while much research has focused on *detecting* inefficient regular expressions or *accelerating* regular expression matching at the hardware level, we investigate automatically *transforming* regular expressions to remove inefficiencies. We reduce this problem to general *expression optimization*, an important task necessary in a variety of domains even beyond compilers, e.g., digital logic design, etc. Syntax-guided synthesis (SyGuS) with a cost function can be used for this purpose, but ordered enumeration through a large space of candidate expressions can be prohibitively expensive. Equality saturation is an alternative approach which allows efficient construction and maintenance of expression equivalence classes generated by rewrite rules, but the procedure may not reach saturation, meaning global minimality cannot be confirmed. We present a new approach called rewrite-guided synthesis (ReGiS), in which a unique interplay between SyGuS and equality saturation-based rewriting helps to overcome these problems, resulting in an efficient, scalable framework for expression optimization.

1 INTRODUCTION

Because regular expressions and their associated operations (matching, etc.) play such a pivotal role in modern systems, there has been much interest in developing hardware acceleration for regular expressions [61, 32, 9, 21, 43, 33]. Our work investigates a complementary approach, namely optimizing regular expressions at the software level. Because there are other popular formalisms that share similar properties to regular expressions (e.g., Boolean algebra), we frame the problem in terms of general expression optimization, enabling straightforward extensions in other domains.

Expression optimization is a type of *program synthesis* problem—we must automatically construct a program (expression) that satisfies some specification (e.g., minimal cost, and equality to the input expression). In the mid-Eighties, Brooks [8] famously identified several technological areas unlikely to result in a "silver bullet" in terms of increased programmer productivity and software quality, and program synthesis appeared in the list. Since then, significant strides have been made in some of these areas, perhaps most notably, data-centric advances in machine learning which have enabled software to perform a variety of complex tasks, including winning chess matches against professionals, driving cars, and landing rockets. Overall, progress in the area of program synthesis has seen more moderate gains. One notable approach is *syntax-guided synthesis* (SyGuS) [2], which has leveraged domain-specific languages (DSLs) and exploited fast solvers (e.g., SAT and SMT [39]) to produce synthesizers usable in areas such as distributed systems [57], robotics [13], biochemical modeling [11], networking [35], and many more. Conceptually, SyGuS performs a search over the space of all program expressions, checking at each step if the expression satisfies the specification. Although various techniques have been devised to make this search more efficient, many of the "big ideas" that have allowed advancement elsewhere (big data, novel hardware processing units, massive parallelization) have proven more difficult to utilize in this type of syntax-guided search.

1.1 Problem Description: Expression Optimization

In this paper, we develop a new optimal synthesis framework called Rewrite-Guided Synthesis (ReGiS) which extends SyGuS, making it more flexible and amenable to parallelization. Our goal is to take an initially-correct *source expression*, and transform it into a *better* equivalent expression. The user can provide the expression language, an optional set of semantics-preserving rewrite rules, a cost metric for expressions, and a source expression, and the synthesizer outputs an equivalent expression that is minimal with respect to the cost metric.

1.2 Existing Approaches

Several existing approaches can be used for expression optimization. *Optimal Synthesis* [7, 11] uses a cost metric and techniques such as counterexample-guided enumeration to search for an optimal program satisfying a specification. *Rewriting* [55, 59] uses syntactic transformations and efficient data structures to produce equivalent expressions with differing structure. *Superoptimization* [47, 44] transforms small snippets of code into equivalent and higher-performing snippets, using enumerative or rewriting-based methods. Section 7 gives more detail about these approaches. In contrast to these, ReGiS uses a unique combination of enumeration and rewriting, resulting in a more flexible and efficient technique.

[*] Equal contribution

PACT '22, October 10–12, 2012, Chicago,IL
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-XXXX-X/18/06.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1.3 ReGiS Novelties

ReGiS targets three core improvements over previous approaches.

(1) Combining enumeration (using semantic correctness/equality) and syntactic rewriting. Enumerative search—symbolically or explicitly iterating through program expressions in increasing order with respect to cost while checking semantic correctness/equality—is often not efficient when the goal is to *optimize* a given input expression in some way, i.e., transform it into an equivalent expression with lower cost. Specifically, since the input expression is already *correct*, it may be counterproductive to "start from scratch" when building an equivalent expression. In many domains, it is possible to find semantics-preserving transformations [12] which allow *rewriting* an expression to obtain lower cost, with respect to a metric like expression size or time complexity. In some cases, these transformation rules have useful properties, e.g., soundness and completeness in the case of Kleene algebra for regular expressions [28], but other times, this is not the case. Thus, just as purely enumerative synthesis has drawbacks, so too does a purely rewrite-based approach, since it requires careful design of the rewrite rules. Additionally, the optimal target expression may have a large distance from the source expression with respect to the rules, and rewriting-based approaches can become rapidly overwhelmed as the search depth increases. For these reasons, we show how to combine enumeration with rewriting, allowing exploration of expressions which are *locally close* (syntactically related) to seen expressions, as well as expressions which are *globally small* (having overall lowest cost).

(2) Using parallelizable bi-directional search. Rather than simply starting from the source expression, and trying to discover a chain of equivalences to a specific target expression, we additionally try to construct these chains *backward* toward the source from several candidate targets simultaneously.

(3) Enabling customizable expression languages and semantics. Our approach is cleanly parameterized over a user-specifiable expression language. While we focus on the domain of regular expressions, the approach would be equally applicable in other domains such as Boolean logic, process algebras, etc.

1.4 ReGiS Approach Overview

ReGiS consists of three components: *Enumerator*, *Updater*, and *Unifier*. The Enumerator iterates through candidate target expressions in increasing order of cost, adding each new candidate to the Updater. When the Updater receives a new target expression, it is added to an efficient *E-graph* data structure [55, 59], allowing *all* known rewrite rules to be applied to the expression, which enables compact maintenance of the *equivalence classes (E-classes)* for the source and candidate target expressions, modulo the known syntactic rewrite rules. The Updater also maintains an *overlay graph* (Figure 1), with E-classes as nodes, and edge labels representing the (initially unit) estimated expense of semantic equality checks between classes. In parallel with these processes, Unifiers systematically attempt to merge E-classes: each Unifier selects a low-cost overlay graph edge, chooses expressions from the two corresponding E-classes, and performs a semantic equality check. If its equality check *succeeds*, it tells the Updater to union the two E-classes, and can potentially provide the Updater with a new rewrite rule(s). If its equality check *fails*, it removes the associated edge. If its equality check *times out*, it increases its edge's expense estimate. Eventually, a target E-class that is minimal with respect to cost will be unioned with the source E-class, allowing ReGiS to terminate and report the global minimum. The current lowest-cost result is available as the minimum-cost expression in the source E-class. In contrast to approaches that extend SyGuS by parallelizing enumeration steps [25], our approach does the syntactic (rewriting-based) and semantic (equality-based) parts of the search in parallel.

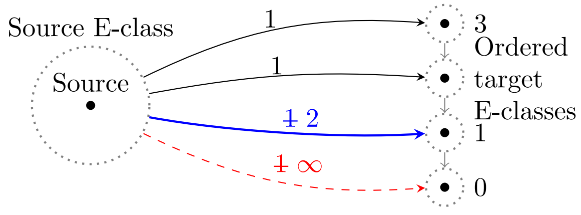

Figure 1: Overlay graph: edge labels encode *expense estimates*; dashed/red edge shows an *inequality* discovered by a Unifier, which causes edge deletion; and thick/blue edge shows a Unifier *timeout*, which increases estimate.

1.5 Paper Organization

This paper is organized as follows: §2 demonstrates why superlinear regular expressions are problematic, and shows how ReGiS can be used to address this; §3 formalizes our approach, and presents correctness results; §4 shows the details of using ReGiS for regular expression optimization; §5 describes our prototype implementation, and provides experimental results; §6 and §7 describe future work and related work; and §8 concludes.

2 MOTIVATING EXAMPLE: OPTIMIZING SUPERLINEAR REGULAR EXPRESSIONS

We demonstrate the utility of our framework by examining problematic behavior of superlinear regular expressions. *Catastrophic backtracking* behavior can be triggered by crafting input strings to target inefficiencies in the regular expression. As an example, consider the regular expression $R_1R_2 = \mathtt{a}^*\mathtt{a}^*$. If we try to match the entire input string $c_1c_2c_3 \ldots c_nc_{n+1} = \mathtt{aaa} \ldots \mathtt{ab}$ using this regular expression, we might first greedily capture $c_1 \ldots c_n$ using R_1, only to realize that there is no way to match the trailing b. We would then need to backtrack and accept $n-1$ leading a characters with R_1, and let R_2 match the final a, which would similarly fail due to the trailing b in the input. This would continue, with R_1 accepting $c_1 \ldots c_k$, and R_2 accepting $c_{k+1} \ldots c_n$, until all k have been tried, resulting in quadratic runtime.

One way to avoid this issue is to use non-backtracking algorithms. For example, we could convert the regular expression to a *nondeterministic finite automaton* (NFA) using Thompson's construction [56], and then determinize the NFA, but this can result in exponential explosion of the automaton size, so this approach is not typically used in practice. Thompson [56] also presented an automaton simulation algorithm which can match a string against

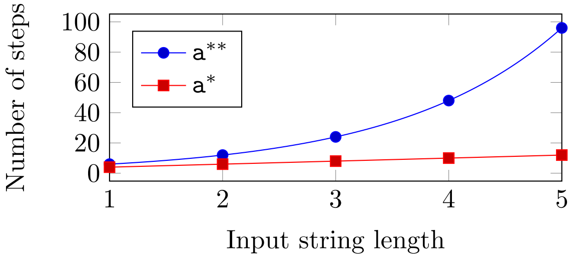

Figure 2: Matching w/ semantically-equivalent expressions (input aa . . . ab).

an NFA in polynomial time. Unfortunately, many real-world regular expression engines have chosen to instead rely on backtracking algorithms, due to complex extensions to the regular expression language (backreferences, etc.). *Perl-Compatible Regular Expressions* (PCRE) is one such implementation [4].

These superlinear regular expressions appear with concerning frequency in real-world systems [53, 19], and real attacks have been observed. As an orthogonal approach to ours, *static analysis* has been used to *detect* exponential regular expressions [45]. Note that focusing on *exponential* regular expressions is insufficient—although the maximum number of operations for backtracking regular expression algorithms is bounded by $2^{\Theta(n)}$ [23, 41], *polynomial* complexity can also be problematic [58].

Several approaches have been identified for dealing with superlinear regular expressions [19], the most promising of which seems to be transforming the expression into an equivalent but less complex one. To our knowledge, however, this has not been solved in a comprehensive way. In this section, we will examine the problem of optimizing superlinear regular expressions in greater detail, and show how the various components of our approach work together to tackle this problem. Consider Figure 2, which shows the performance of the standard PCRE matching algorithm for the regular expressions a^* and a^{**}. These regular expressions are *semantically equivalent*, i.e., they recognize the same language, but their differing syntactic structures cause drastically different performance when matching the previously-described input string aa . . . ab. Regular expression a^* has linear performance $\Theta(n)$, while a^{**} has exponential performance $\Theta(2^n)$, and each additional added star increases the base of the exponent. Intuitively, at each step, a^* has only two options: accept a *single* a character or fail on the trailing b character, but a^{**} can accept an arbitrarily-long *sequence* of a characters at each step, forcing the algorithm to try all possible combinations of sequence lengths before failing. In Sec. 4, we cover this example in more detail, and introduce a cost metric that characterizes such backtracking behavior.

2.1 Limitations of Basic Rewriting

One basic optimization approach is to perform rewriting using the well-known Kleene algebra axioms [28], at each step checking whether we have found an expression that has lower cost according to our metric. For example, given a + a + a (where + denotes alternation), we can use the idempotence rule $x{+}x \longleftrightarrow x$ to perform the rewrites a+a+a $\longrightarrow$ a+a $\longrightarrow$ a, and we will have reached an equivalent regular expression with lower cost.

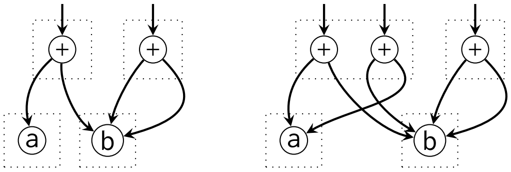

Figure 3: (a) E-graph initially built from a + b and b + b, and (b) after equality saturation using rewrite rule $x + y \longleftrightarrow y + x$.

This basic approach scales poorly—in general, we would need to perform a rewrite-based search, iterating through the various rewrite rules. The search can "loop", e.g., rewriting an expression into progressively larger expressions. Note that we cannot restrict rewrites to only *shrink* expression cost, because in some cases, *global* minimization necessitates local monotonic (or even increasing) rewrites during the search. As an example, optimizing $1 + a^*$ (where 1 denotes the empty string) requires a rewrite which initially increases cost. Specifically, using arrow angle to indicate change in cost due to a rewrite, we have $1{+}a^* \nearrow 1 + 1 + aa^* \searrow 1{+}aa^* \searrow a^*$. Regular expression optimizers based on this type of rewrite-based search often timeout before making any progress. For regular expressions such as a + b + c + d + e + d + c + b + a (which is clearly reducible to a+b+c+d+e), the search would need to conceptually "sort" the characters using commutativity of alternation, and then use idempotence, requiring a huge amount of search.

2.2 Limitations of E-Graph-based Rewriting

Equality saturation is a technique for efficiently implementing a rewriting-based task such as the one previously described. This approach uses a data structure called an E-graph to compactly store one or more initial expressions, along with expressions derivable from these via a set of rewrite rules. Figure 3(a) shows an example, namely the E-graph containing regular expressions a + b and b + b. Equality saturation can apply the *commutativity* rewrite rule, which adds the expression b + a to the E-graph, resulting in Figure 3(b). Note that each subexpression a and b is stored only once—the E-graph maintains this type of expression sharing to keep the size compact. The dotted boxes in the figure represent E-classes—equivalence classes with respect to the rewrite rules. Expressions a + b and b + a are in the same E-class, since they are equivalent with respect to the rewrite rule, but b + b is in a separate E-class.

With an E-graph-based rewriting approach, the straightforward way to implement regular expression optimization is to first add the source expression to the E-graph, run equality saturation using all of the Kleene algebra axioms as rewrite rules, and iterate over the source regular expression's E-class to find the minimal equivalent expression with respect to the cost metric. There are two key problems with this. (1) Although *cyclic* edges in the E-graph can sometimes be used to encode infinite sets, in general, equality saturation may not have enough time or resources to fully *saturate* the E-graph in cases where there are infinitely many equivalent expressions with respect to the Kleene algebra axioms (e.g., a = a + a = a + a + a = · · ·), meaning the procedure may need to time out. (2) ReGiS is designed to be *general*, and in some cases, we may have a more limited set of rewrite rules—in particular, we may not have a *completeness* result, meaning that for some semantically equivalent expressions, it may not be possible to show their equivalence using the syntactic rewrite rules alone.

As an example, consider optimizing $(1 + \mathsf{a}^*\mathsf{a})^{**}$, using only two rewrite rules: $1 + xx^* \overset{1}{\longleftrightarrow} x^*$, $x^{**} \overset{2}{\longleftrightarrow} x^*$. What we would need is a chain of rewrites:

$$(1 + \mathsf{a}^*\mathsf{a})^{**} \overset{2}{\longrightarrow} (1 + \mathsf{a}^*\mathsf{a})^* \overset{?}{\longrightarrow} (1 + \mathsf{aa}^*)^* \overset{1}{\longrightarrow} \mathsf{a}^{**} \overset{2}{\longrightarrow} \mathsf{a}^*$$

Here, it is not possible to build this chain of equalities using the available syntactic rewrite rules, so we would need a *semantic* equality check to "bridge the gap" between $(1+\mathsf{a}^*\mathsf{a})^*$ and $(1+\mathsf{aa}^*)^*$.

2.3 Enumerative Bidirectional Rewriting

This is the basic idea of our enumerative bidirectional rewriting approach. We use a SyGuS-based strategy to enumerate candidate target regexes by increasing cost, and adding them to the E-graph. Equality saturation applies rewrites to the source and all targets simultaneously. For any target whose E-class intersects the source's E-class, the E-graph will *union* these E-classes. We iterate through E-classes which are currently disjoint but potentially equal, and try to equate these using a semantic equality check (NFA bisimilarity). In this example, a successful equality check $\mathsf{a}^*\mathsf{a} = \mathsf{aa}^*$ could result in a new rewrite rule $a^*a \leftrightarrow aa^*$, allowing equality saturation to "bridge the gap" indicated by "$\overset{?}{\longrightarrow}$".

3 REGIS: REWRITE-GUIDED SYNTHESIS

In this section, we formalize our rewrite-guided synthesis approach, and describe key properties of the algorithm. In Section 4, we show in detail how our approach can be used to tackle the real-world problem of optimizing superlinear regular expressions.

3.1 Expression Optimization

We first specify the problem statement. Let G be a grammar, and let $\mathcal{E} = L(G)$ be G's *language*, i.e., the set of expressions that can be built from G. Let *height* : $\mathcal{E} \longrightarrow \mathbb{N}$ denote height of an expression's tree. Let *subexprs* : $\mathcal{E} \longrightarrow \mathcal{P}(\mathcal{E})$ denote subexpressions.

Let *cost* : $\mathcal{E} \longrightarrow \mathbb{R}$ be a *cost function* that assigns a numeric cost to each expression. Let $[\![\cdot]\!] : \mathcal{E} \longrightarrow D$ denote the *semantics* of the expression language, i.e., a function that maps expressions to objects of some domain D, and let $\approx : (D \times D) \longrightarrow \mathbb{B}$ be a *semantic equality* function for comparing objects in that domain. Let *hl* : $\mathcal{E} \longrightarrow \mathcal{P}(\mathcal{E})$ denote equivalent expressions of equal or lesser height, i.e., $hl(e) = \{e' \in \mathcal{E} \mid height(e') \leq height(e) \text{ and } [\![e']\!] \approx [\![e]\!]\}$. Given a grammar G, we define a *pattern* to be an expression initially built from G, where zero or more subexpressions have been replaced with *variables* from a set V. Intuitively, variables serve as placeholders for arbitrary subexpressions built from G. If p is a pattern, and $m : V \longrightarrow \mathcal{E}$ is a mapping, we use $p[m]$ to denote the expression formed by applying m to each variable in p. Note that if p contains no variables, $p[m] = p$ for all m, and if $\emptyset$ denotes the empty map, $p[\emptyset] = p$ for all p. We define a *rewrite rule* to be an object of the form $p_1 \longrightarrow p_2$, where p_1, p_2 are patterns, and a *bidirectional* rewrite rule to be of the form $p_1 \leftrightarrow p_2$. We say rewrite rule $p_1 \longrightarrow p_2$ *matches* e if and only if there is a mapping $m : V \longrightarrow \mathcal{E}$ such that $p_1[m] = e$, and in this case, we say that $rewrite(e, p_1 \longrightarrow p_2) = \{p_2[m]\}$. If e does not match p_1, then $rewrite(e, p_1 \longrightarrow p_2) = \emptyset$. If E is a set of expressions and W is a set of rewrite rules, $rewrite(E, W)$ signifies $\bigcup_{e \in E,\, w \in W} rewrite(e, w)$.

Let W be a *sound* set of rewrite rules, i.e., for any $w \in W$, if $e' \in rewrite(e, w)$, then $[\![e']\!] \approx [\![e]\!]$. An *optimization instance* is a tuple $(e, cost, W, [\![\cdot]\!], \approx)$ where $e \in \mathcal{E}$, and the *optimization* problem consists of finding a *minimal* equivalent expression, i.e., an $e' \in \mathcal{E}$ such that $[\![e']\!] \approx [\![e]\!]$ and for any e'' where $[\![e'']\!] \approx [\![e]\!]$, we must have $cost(e') \leq cost(e'')$.

3.2 E-Graphs

Given an optimization instance, we encode the expression language $\mathcal{E}$ using the equality saturation framework Egg [59], which accepts a straightforward s-expression-based formulation of the grammar. Although Egg contains significant machinery to ensure that E-graphs are maintained compactly, for our formalization purposes, we consider an *E-graph* to be a mapping of the form $E : \mathcal{E} \longrightarrow (\mathbb{N} \times \mathcal{E})$, i.e., each contained expression e within E-graph E is associated with a numeric E-class identifier $E_{id}(e)$ and the minimum-cost expression $E_{min}(e)$ within that E-class. We use $class(E, e)$ to denote the set of all expressions contained in the same E-class as e.

3.3 ReGiS Algorithm

Figure 4 formalizes ReGiS as an *abstract machine* [5]. A rule of the form $\frac{C}{S \longrightarrow S'}$ can be applied to step the machine state from S to S' if the condition C is satisfied. The algorithm terminates when no further steps can be taken. A machine state is of the form $\langle X, E, O, W, U, k \rangle$, where X is a set used for storing the global minimum (return value); E is the E-graph; O is a tuple (D, S, T) representing the overlay graph, where S and T are the lists of source/target expressions respectively (overlay graph nodes), and D is a set of weighted overlay graph edges; W is the set of rewrite rules; U is a set of expression (in)equalities to be incorporated into the E-graph; and k is the index of the minimal unprocessed target in T, i.e., lowest-cost target that has not yet been (in)equality-checked against the source (in the Figure 1 example, this would be the target at index 1). Given optimization instance $(e, cost, W, [\![\cdot]\!], \approx)$, we use initial machine state $\langle \emptyset, E, (\emptyset, [e], \emptyset), W, \emptyset, 0 \rangle$, and run the machine until X becomes non-empty, which causes the machine to halt (the expression contained in X is the global minimum returned by the algorithm). If the user prematurely terminates the machine, we can output the current minimum $E_{min}(S_0)$, which may have lower cost than the source expression S_0, but may not yet be the global minimum.

3.4 Updater

The Updater's functionality is described in Figure 4 by the Rewrite, Saturate, and Union$X$ rules. Conceptually, the Updater functions as a wrapper for a persistent instance of Egg's E-graph data structure, which is denoted E. Rewrite allows a single rewrite rule w that matches an expression e to be applied, and adds the resulting equality $e{=}e'$ to the set U to be incorporated into the E-graph via the Union$X$ rules. Saturate is for cases where the rewrite rules have a completeness result. This rule tests for a *saturated* E-graph, i.e., in which none of the rewrite rules change E—in this case, the algorithm can terminate (returning the global minimum $E_{min}(S_0)$),

Updater

$$\dfrac{w \in W \quad e \in E \quad e' \in \mathit{rewrite}(e, w) \quad e' \notin E}{\langle \emptyset, E, O, W, U, k \rangle \longrightarrow \langle \emptyset, E, O, W, U \cup \{e{=}e'\}, k \rangle}\ \textsc{Rewrite}$$

$$\dfrac{\mathit{rewrite}(E, W) \subseteq E}{\langle \emptyset, E, O, W, U, k \rangle \longrightarrow \langle \{E_{min}(S_0)\}, E, O, W, U, k \rangle}\ \textsc{Saturate}^{\dagger}$$

$$\dfrac{\begin{array}{c} U = U' \cup \{e{=}e'\} \quad E = E' \cup \{e \mapsto (i, m), e' \mapsto (i', m')\} \quad E_{in} = \{e_s \mid (e_s \to e') \in D\} \quad E_{out} = \{e_t \mid (e' \to e_t) \in D\} \quad D_{in} = \{(e_s{\to}e') \mid e_s{\in}E_{in}\} \\ D_{out} = \{(e'{\to}e_t) \mid e_t{\in}E_{out}\} \quad D = D'{\cup}D_{in}{\cup}D_{out} \quad I_s = \{k \mid E_{id}(S_k) \in \{E_{id}(e), E_{id}(e')\}\} \quad I_t = \{k \mid E_{id}(T_k) \in \{E_{id}(e), E_{id}(e')\}\} \end{array}}{\begin{array}{c} \langle \emptyset, E, (D, S, T), W, U, k \rangle \longrightarrow \langle \emptyset, E'{\cup}\{e{\mapsto}(i, \mathit{min}(m, m')), e'{\mapsto}(i, \mathit{min}(m, m'))\}, \quad (D'{\cup}\{e_s{\to}e \mid e_s{\in}E_{in}\}{\cup}\{e{\to}e_t \mid e_t{\in}E_{out}\}, \\ S{-}(I_s{-}\mathit{min}(I_s)), T{-}(I_t{-}\mathit{min}(I_t))), W, U', k \rangle \end{array}}\ \textsc{Un.1}$$

$$\dfrac{E_{id}(S_0) = E_{id}(T_k)}{\langle \emptyset, E, O, W, U, k \rangle \longrightarrow \langle E_{min}(S_0), E, O, W, U', k \rangle}\ \textsc{Union2}$$

$$\dfrac{\begin{array}{c} U{=}U'{\cup}\{e{\neq}e'\} \\ D{=}D' \cup \{e_s \to e_t \mid \{E_{id}(e), E_{id}(e')\} = \{E_{id}(e_s), E_{id}(e_t)\}\} \qquad 1 \geq n \geq 0 \quad \{E_{id}(e), E_{id}(e')\}{=}\{E_{id}(S_0), E_{id}(T_k)\} \iff n{=}1 \end{array}}{\langle \emptyset, E, (D, S, T), W, U, k \rangle \longrightarrow \langle \emptyset, E, (D', S, T), W, U', k + n \rangle}\ \textsc{Un.3}$$

Unifier

$$\dfrac{T = [\ldots, e_t, \ldots] \quad D = D' \cup \{e_s \xrightarrow{c} e_t\} \quad c \leq \mathit{min}(\{j \mid (x \xrightarrow{j} y) \in D'\}) \quad e \in \mathit{class}(E, e_s) \quad e' \in \mathit{class}(E, e_t) \quad [\![e]\!] \approx [\![e']\!]}{\langle \emptyset, E, (D, S, T), W, U, k \rangle \longrightarrow \langle \emptyset, E, (D', S, T), W \cup \{e \leftrightarrow e'\}, U \cup \{e{=}e'\}, k \rangle}\ \textsc{Eq.}$$

$$\dfrac{T = [\ldots, e_t, \ldots] \quad D = D' \cup \{e_s \xrightarrow{c} e_t\} \quad c \leq \mathit{min}(\{j \mid (x \xrightarrow{j} y) \in D'\}) \quad e \in \mathit{class}(E, e_s) \quad e' \in \mathit{class}(E, e_t) \quad [\![e]\!] \not\approx [\![e']\!]}{\langle \emptyset, E, (D, S, T), W, U, k \rangle \longrightarrow \langle \emptyset, E, (D', S, T), W, U \cup \{e{\neq}e'\}, k \rangle}\ \textsc{Ineq.}$$

$$\dfrac{T = [\ldots, e_t, \ldots] \quad D = D' \cup \{e_s \xrightarrow{c} e_t\} \quad c \leq \mathit{min}(\{j \mid (x \xrightarrow{j} y) \in D'\}) \quad e \in \mathit{class}(E, e_s) \quad e' \in \mathit{class}(E, e_t) \quad [\![e]\!] \mathrel{?{\approx}} [\![e']\!]}{\langle \emptyset, E, (D, S, T), W, U, k \rangle \longrightarrow \langle \emptyset, E, (D' \cup \{e_s \xrightarrow{2 \cdot c} e_t\}, S, T), W, U, k \rangle}\ \textsc{Timeout}$$

Enumerator

$$\dfrac{T = [\ldots, e_n] \quad e \in \mathit{min}(\{e' \mid \mathit{cost}(E_{min}(S_0)) \geq \mathit{cost}(e') \geq \mathit{cost}(e_n)\})}{\langle \emptyset, E, (D, S, T), W, U, k \rangle \longrightarrow \langle \emptyset, E, (D \cup \{e_s \xrightarrow{1} e' \mid e_s \in S\}, S, T{+}[e]), W, U, k \rangle}\ \textsc{Enum.}$$

$$\dfrac{S = [\ldots, e', \ldots] \quad e \in \mathit{subexprs}(e')}{\langle \emptyset, E, (D, S, T), W, U, k \rangle \longrightarrow \langle \emptyset, E, (D \cup \{e \xrightarrow{1} e_t \mid e_t \in T\}, S{+}[e], T), W, U, k \rangle}\ \textsc{Source}$$

Figure 4: ReGiS as an abstract machine with ⟶ denoting transitions. Machine state is $\langle X, E, O, W, U, k \rangle$: X is a set containing a minimal regular expression upon termination; E is the E-graph; $O = (D, S, T)$ is the overlay graph with set of edges D, list of sources S, and list of targets T; W is the set of rewrite rules; U is a set of (in)equalities to be processed; and k is the index of the minimal unprocessed target in T. †

SATURATE is used only in contexts where the rewrite rules have a completeness result.

since all possible rewrites have been explored. The UNION2 rule allows the algorithm to terminate when the minimal unprocessed target T_k has been added to the source expression's E-class, since the enumeration order ensures that T_k will contain a globally-minimal expression. UNION1 incorporates an equality $e{=}e'$ into the E-graph E, by (1) placing e' into e's E-class within E, (2) updating the overlay graph edges D by moving incoming/outgoing edges from e' to e, and (3) updating the overlay graph source/target lists by keeping only the *lowest* index belonging to the same E-class as e or e', ensuring that the source (S_0) and minimal unprocessed target (T_k) expressions are not absorbed into other sources/targets. UNION3 incorporates an inequality $e{\neq}e'$ obtained from the Unifier, by deleting any corresponding edges from the overlay graph. If e and e' are contained in the source (S_0) and minimal unprocessed target (T_k) E-classes respectively (or vice versa), UNION3 additionally increments the index of the minimal unprocessed target T_k.

3.5 Unifier

Each spawned Unifier selects a minimal-weight source/target edge $e_s \xrightarrow{c} e_t$ from the overlay graph, and performs a semantic equality check, as shown in EQUALITY, INEQUALITY, and TIMEOUT in Figure 4. A single member from each class is selected (e and e'), and the equality check $[\![e]\!] \approx [\![e']\!]$ is performed. If the check *succeeds* (EQUALITY), the Unifier records e and e' as needing to be unioned, and a new rewrite rule $e \leftrightarrow e'$ is generated. If the check *fails* (INEQUALITY), the Unifier records inequality $e{\neq}e'$, which the Updater will use to delete the overlay graph edge, ensuring that this particular equality check is not attempted again. If the equality-checking procedure *times out* (TIMEOUT), the Unifier increases the expense estimate of the edge joining e_s and e_t, ensuring that other potentially-easier equality checks are tried before returning to this pair. Intuitively, timeout of the equality check means that we do not (yet) know whether the two expressions are semantically equivalent.

3.6 Enumerator

The Enumerator's goal is to iterate through expressions in order of increasing cost, and add them as targets (along with new overlay graph edges), as shown in the Enumerate rule. One key issue with in-order enumeration is the sheer number of expressions involved, which can be in the millions for even depth-4 binary trees. This causes slowdown of the Enumerator itself, and makes it unlikely that higher-cost expressions will be reached in a reasonable amount of time. Rewriting helps address this problem, by maintaining the *minimum expression* $E_{min}(S_0)$ in the source expression's E-class. As this expression's cost gets reduced by rewriting, $cost(E_{min}(S_0))$ becomes the new *upper bound* for cost within the Enumerator, reducing the search space, and speeding up enumeration.

Our Enumerator is powered by the Z3 SMT solver [36]. We require the cost function to be representable in SMT using a decidable theory such as QF_UFLIA (quantifier-free uninterpreted function symbols and linear integer arithmetic), and we use uninterpreted function symbols to encode expressions as trees up to a specified maximum height, representing the cost metric via assertions that maintain the cost of each tree node. Section 4.6 contains encoding details for regular expressions.

In addition to adding targets, the Enumerator can also add additional source expressions (Source rule), to facilitate equality checking against *subexpressions* of the source.

3.7 Properties of ReGiS

Theorem 3.1 (Soundness). *If ReGiS returns an expression e' for an optimization instance $(e, cost, W, [\![\cdot]\!], \approx)$, then e' is no larger than the minimal expression in $hl(e)$, i.e., $cost(e') \leq cost(e'')$ for any $e'' \in hl(e)$.*

Theorem 3.2 (Completeness). *If e_m is a minimal expression in $hl(e)$, i.e., $e_m \in hl(e)$ and $cost(e_m) \leq cost(e'')$ for any $e'' \in hl(e)$, then ReGiS's result e' will have $cost(e') = cost(e_m)$.*

The proofs of these theorems appear in Appendix A.

4 REGULAR EXPRESSION OPTIMIZATION

In Section 3, we formalized the ReGiS framework, and in this section, we highlight the flexibility and practicality of our approach by using it to solve the important and insufficiently-addressed problem of optimizing superlinear regular expressions.

4.1 Regular Expression Preliminaries

While §3.1 discussed expressions generally, we will now focus specifically on *regular expressions*. Regular expressions are a classic formalism providing a compositional syntactic approach for describing *regular languages*, and are useful for tokenizing input streams (e.g., in a lexer), pattern matching within text, etc. In software development practice, the term "regular expression" is often overloaded to refer to a variety of pattern-matching capabilities and syntaxes, so it is important to fix this definition for our work. We say that an expression R is a *regular expression* if and only if it matches the following grammar (we will use the term *regex* when referring to pattern-matching expressions beyond this core language).

$$\begin{array}{lcll} R & ::= & 0 \mid 1 \mid c \mid R + R \mid R \cdot R \mid R^* & \textit{(regular expression)} \\ c & \in & \mathcal{A} & \textit{(character from alphabet)} \end{array}$$

An expression $R_1{\cdot}R_2$ is often written as R_1R_2. Semantics can be defined in terms of the language each regular expression recognizes.

$$\begin{array}{lcl} L(0) & = & \emptyset \\ L(1) & = & \{\epsilon\} \\ L(c) & = & \{c\} \\ L(R_1 + R_2) & = & L(R_1) \cup L(R_2) \\ L(R_1 \cdot R_2) & = & \{s_1 s_2 \mid s_1 \in L(R_1) \text{ and } s_2 \in L(R_2)\} \\ L(R^*) & = & \bigcup_{k=0}^{\infty} L(R^k) \end{array}$$

We use R^k to mean $\overbrace{R{\cdot}R{\cdot}R{\cdot}\cdots}^{k}$ for $k > 0$, and $R^0 = 1$. The above semantics tells us that 0 recognizes no strings, 1 recognizes the empty string ϵ, character c recognizes the corresponding single-character string, alternation $+$ recognizes the union of two languages, concatenation $\cdot$ recognizes string concatenation, and iteration (Kleene star) $*$ recognizes repeated concatenation.

4.2 Regular Expression Semantic Equality

To perform the Section 3.5 (Unifier) semantic equality check $\approx$ for regular expressions R_1 and R_2, we must decide whether $L(R_1) = L(R_2)$, i.e., whether they describe the same language. Using Thompson's construction, we can efficiently convert R to an NFA $N(R)$ which recognizes the language $L(R)$, and this result allows us to instead focus on the equality check $L(N(R_1)) = L(N(R_2))$. NFA equality is PSPACE-complete [34], but we utilize a bisimulation-based algorithm which has been shown to be effective in many cases [1, 20].

4.3 Regular Expression Syntactic Rewriting

Regular expressions have a mathematical formalization known as *Kleene algebra*. Due to soundness and completeness properties, equality of regular expressions can be fully characterized by a set of Kleene algebra axioms. These axioms leave us with two equally powerful ways checking equality of regular expressions R_1, R_2: we can either check whether they describe the same language, i.e., $L(R_1) = L(R_2)$ (semantic equality) as discussed in Section 4.2, or we can check whether there is a *proof* using the Kleene algebra axioms showing that $R_1 = R_2$ (syntactic equality).

Following Kozen [28], we list the Kleene algebra axioms as follows, where $R_1 \leq R_2$ is shorthand for $R_1 + R_2 = R_2$.

$$\begin{array}{rcll} A + (B + C) & = & (A + B) + C & \text{associativity of } + \quad (1) \\ A + B & = & B + A & \text{commutativity of } + \quad (2) \\ A + 0 & = & A & \text{identity for } + \quad (3) \\ A + A & = & A & \text{idempotence of } + \quad (4) \\ A \cdot (B \cdot C) & = & (A \cdot B) \cdot C & \text{associativity of } \cdot \quad (5) \\ 1 \cdot A & = & A & \text{left identity for } \cdot \quad (6) \\ A \cdot 1 & = & A & \text{right identity for } \cdot \quad (7) \\ A \cdot (B + C) & = & A{\cdot}B + A{\cdot}C & \text{left distributivity of } \cdot \quad (8) \\ (A + B) \cdot C & = & A{\cdot}C + B{\cdot}C & \text{right distributivity of } \cdot \quad (9) \\ 0 \cdot A & = & 0 & \text{left annihilator for } \cdot \quad (10) \\ A \cdot 0 & = & 0 & \text{right annihilator for } \cdot \quad (11) \\ 1 + A{\cdot}A^* & \leq & A^* & \text{left unrolling of } * \quad (12) \\ 1 + A^*{\cdot}A & \leq & A^* & \text{right unrolling of } * \quad (13) \\ B + A{\cdot}X \leq X & \Rightarrow & A^*{\cdot}B \leq X & \text{left unrolling ineq.} \quad (14) \\ B + X{\cdot}A \leq X & \Rightarrow & B{\cdot}A^* \leq X & \text{right unrolling ineq.} \quad (15) \end{array}$$

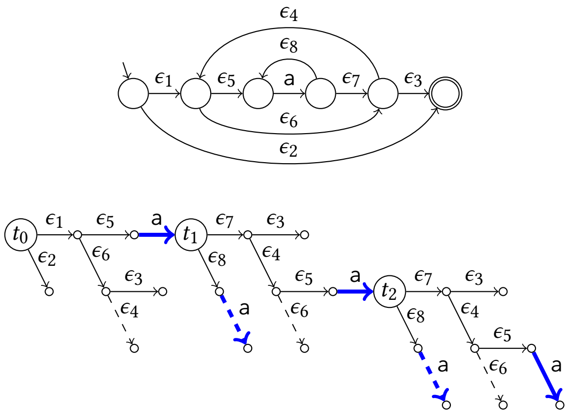

Figure 5: (a) NFA for regular expression a^{**}**, and (b) its (partial) computation tree.**

We can derive the following useful rules from the above axioms. These rules are extraneous due to the soundness and completeness of 1-15, but they offer several ways of quickly eliminating Kleene stars, which as seen in Section 4.5, are a primary contributor toward superlinear behavior.

$$\begin{array}{rcllr} 1 + A{\cdot}A^* & = & A^* & \text{strong left unrolling of } * & (16)\\ 1 + A^*{\cdot}A & = & A^* & \text{strong right unrolling of } * & (17)\\ A^* \cdot A^* & = & A^* & \text{idempotence of } * & (18)\\ A^{**} & = & A^* & \text{saturation of } * & (19)\\ 1^* & = & 1 & \text{iterated identity} & (20)\\ 0^* & = & 1 & \text{iterated annihilator} & (21) \end{array}$$

We encode equations 1-11, 16-17 (the stronger forms of 12-13), and 18-21 as rewrite rules—for each equation $X = Y$, we produce the bidirectional rewrite rule $X \leftrightarrow Y$, and instantiate our Updater (3.4) with these rules. Note that equations 14-15 are not equalities like the others, meaning they are not readily usable as rewrite rules. We omit these equations, noting that the ReGiS approach fully supports incomplete sets of rewrite rules.

4.4 Backtracking Regular Expression Matching Algorithms

Catastrophic backtracking behavior arises due to the *nondeterminism* in the NFAs corresponding to regular expressions. Figure 2 showed a quantitative example of the PCRE engine's exponential behavior on the regular expression a^{**} and input strings aa . . . ab. In Figure 5, we visualize how this occurs, by examining the paths taken through NFA $N(\mathsf{a}^{**})$ as the PCRE matching algorithm attempts to match, and observing how the number of iterations increases exponentially. To match against the string b, all transitions to the *left* of computation tree node t_1 in Figure 5(b) must be explored, before concluding that b is not accepted (7 transitions). Note that backtracking algorithms use lightweight *memoization* to handle cycles in the NFA (skipped transitions due to this behavior are indicated with dashed lines) [16]. To match against the string ab, all transitions to the left of node t_2 must be explored (15 transitions), and to match against aab, all transitions in Figure 5(b) must be explored (23 transitions). Here, superlinear behavior can be triggered by a large sequence of repeated a characters followed by a non-a character. In practice, one can automatically derive such an *attack string* to exploit a given superlinear regular expression [48].

4.5 Cost Metric for Superlinear Regular Expressions

An upper bound on the maximum backtracking for a given regular expression can be characterized by *tree width (leaf size)* [41, 10], which describes the number of leaves in the tree consisting of all possible paths through the regular expression's NFA, but this is hard to compute (PSPACE-complete) [10]. In Figure 5, tree width would be 10 with respect to the input string aab, since this (depth-3) computation tree has 10 leaves. Alternative metrics such as maximal backtracking run [58] quantify potential backtracking in different ways, but are also computationally expensive. We introduce a useful cost metric which we call *backtracking factor* that is quick to compute directly on a regular expression, yet still captures the key syntactic features causing superlinearity in backtracking search. This backtracking factor allows ordering of regular expressions according to "degree of superlinearity", e.g., $cost(\mathsf{a}) < cost(\mathsf{a}^*) < cost(\mathsf{a}^{**})$.

The following shows our cost metric, where $K_1 = |\mathcal{A}| \times (2^h - 1)$ and $K_2 = {K_1}^h \times (K_1 + 2)$ are integer scaling factors for $\cdot$ and $*$ respectively, h is the maximum expression height being used in the Enumerator, and $|\mathcal{A}|$ is the size of the regular expression's alphabet.

$$\begin{array}{lcl} cost(0) & = & 1\\ cost(1) & = & 1\\ cost(c) & = & 1\\ cost(R_1 + R_2) & = & cost(R_1) + cost(R_2)\\ cost(R_1 \cdot R_2) & = & K_1 \times (cost(R_1) + cost(R_2))\\ cost(R^*) & = & K_2 \times cost(R) \end{array}$$

This metric has the effect of ensuring that, for regular expressions up to a maximum height h, the + operator increases cost *additively*, the $\cdot$ operator increases cost *multiplicatively*, and $*$ increases cost *exponentially*. We instantiate our Enumerator (Section 3.6) with this specific cost metric, and in Section 5, we experimentally validate the quality of this metric for characterizing superlinear behavior.

The following property of *cost* says that for the lowest-cost expression R_2 seen so far, the regular expression with *globally* minimal cost will have height no greater than $height(R_2)$. This ensures that we can soundly reduce the Enumerator's *height* bound based on the height of the current lowest-cost expression in the source expression's E-class, which improves enumeration performance.

Theorem 4.1 (Height vs. Cost). *Consider regular expressions* R_1, R_2*. If* $L(R_1) = L(R_2)$*,* $height(R_1) > height(R_2)$*, and* $cost(R_1) \leq cost(R_2)$*, then* $\exists R'$ *such that* $L(R') = L(R_2)$*,* $height(R') \leq height(R_2)$*, and* $cost(R') \leq cost(R_1)$*.*

4.6 SMT Implementation of Cost Metric

We allow the cost metric *cost* to be specified as a recursive integer-valued function using addition as well as multiplication by integer constants (the Section 4.5 cost metric is of this form). This allows us to implement *cost* in Egg as a recursive Rust function, used for

$$\begin{array}{rcl} & \vdots & \\ cost(c) & = & 1 \\ cost(R_1 + R_2) & = & cost(R_1) + cost(R_2) \\ & \vdots & \end{array}$$

Figure 6: Example cost function.

```
(assert (= (ncost N)
           (ite (= (ntype N) CHAR) 1
           (ite (= (ntype N) PLUS) (+ (ncost (left N)) (ncost (right N)))
           ...))))
```

Figure 7: SMT encoding.

extracting the minimal expression from an E-class. Additionally, this allows us to implement *cost* in SMT using the QF_UFLIA theory, to enable enumerating regular expressions by increasing cost, as needed by the Enumerator (Section 3.6).

Our SMT encoding uses uninterpreted function symbols *ntype* : $\mathbb{N} \rightarrow \mathbb{N}$ and *ncost* : $\mathbb{N} \rightarrow \mathbb{N}$, representing the type and cost of each node in the expression's tree. Figures 6-7 demonstrate how our regular expression cost function is encoded using these function symbols. The uppercase symbols in the SMT encoding signify integer constants—e.g., *CHAR* identifies a character expression node type, and *PLUS* identifies an alternation expression node type. We generate one such assertion for each node index N, up to the bounds given by the source expression's height. Integer constants (*left* N) and (*right* N) are the indices of the nodes corresponding to node N's left and right subexpressions respectively.

In the model obtained from the solver, the *ntype* function symbol encodes the expression itself, and (*ncost* 0) contains the expression's cost (index 0 corresponds to the expression's root node).

5 PROTOTYPE AND EVALUATION

We built a prototype of ReGiS (§3), and leveraged it to build a regular expression optimization system (§4) using ~7500 lines of Rust code and several hundred lines of Python/shell script.

The platform used for all experiments was a Dell OptiPlex 7080 workstation running Ubuntu 18.04.2, with a 10-core (20-thread) Intel i9-10900K CPU (3.70GHz), 128 GB DDR4 RAM, and a 2TB PCIe NVME Class 40 SSD. We examine three key research questions Q1-3, to understand the performance and usability of our approach. Note that we have proven the correctness of our algorithm (Section 3.7), but to confirm that the implementation is bug-free, we used the equality checking procedure (Section 4.2) to successfully check the correctness of each result produced by our tool in the experiments.

Q1: How does ReGiS compare against SyGuS and rewriting?

We demonstrate the benefits of *combining* enumerative synthesis and rewriting, by comparing ReGiS performance against each of these approaches operating on their own. We used the Enumerator to simply iterate through candidate expressions in increasing order of cost, performing an NFA equality check against the source expression for each candidate. The input source expressions consisted of all possible regular expressions with a single-character alphabet, up to height 3, for a total of 2777 inputs. Figure 8 shows these performances results.

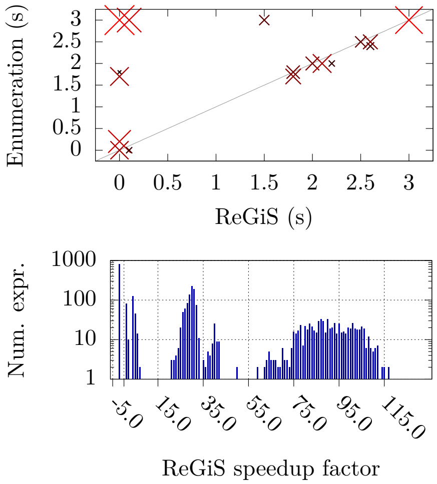

Figure 8: Performance of ReGiS against basic bottom-up enumeration.

Figure 8(a) contains many overlapping points, so we binned the data (bin size 0.1) and used size/color of the points to indicate relative numbers of tests appearing at those locations. The diagonal indicates 1x speedup, so points appearing *above* this line indicate better performance for ReGiS versus basic enumeration. Figure 8(b) visualizes the data differently, showing the spectrum of *speedups* offered by ReGiS. Each bar represents the number of tests in which ReGiS had the speedup shown on the x axis, e.g., 222 tests had a speedup of 30x.

The bar(s) to the left of $x = 0$ contain 821 examples. Of these, 644 are due to timeout of both *both* ReGiS and basic enumeration (time limit 3s), and the remaining 177 were instances where ReGiS was slower. Of these, the average slowdown was 1.06x, and the maximum slowdown was 2x. Only 10 cases were worse than 1.15x slowdown, and in each of these cases, the total runtime of ReGiS was less than 110ms. We are confident that if the timeout were increased slightly, many of the 644 examples would show speedup for ReGiS.

We also added an Egg-only mode to enable rewriting without enumeration. Here, we simply added the input source expression to the E-graph, and applied rewrites until Egg indicated saturation had been reached. All of the input regular expressions timed out at 3s using this rewriting-only mode.

Q2: How much does the *interplay* between enumeration and rewriting help?

One key question is whether enumeration and rewriting could be decoupled while still obtaining the same results. For example, we could first let Egg perform some rewriting, and after we notice that no further reduction in cost seems to be occurring, terminate Egg, and begin enumerating based on the best-cost expression so far. If the performance of this approach were comparable to ReGiS, that would mean our work's unique interaction between rewriting and synthesis may be less important than expected. We set up a simple experiment similar to the alternation example described in Section 2.1, consisting of depth-4 regular expressions each having

Table 1: Reduction in needed semantic equality checks.

	ReGiS		Enumeration	
	Checks	Runtime (s)	Checks	Runtime (s)
Min.	3067	11.80	3416	11.09
Mean	3134.44	12.11	3416	11.40
Max.	3235	12.33	3416	11.86

5 distinct characters, and using only alternation. In this case, (1) the expressions were not reducible, i.e., we must always enumerate up to the maximum depth to confirm the global minimum has been found, and (2) there were many *equivalent* expressions involved in enumeration, due to commutativity of alternation.

Table 1 shows the results on a benchmark set of 100 depth-4 regular expressions. There was reduction in the number of *equality checks* needed by ReGiS—this is because the E-graph is continuously unioning candidate expressions added from the Enumerator, meaning that by the time a Unifier would be spawned for a given pair of E-classes, they may have already been handled by rewriting. Total ReGiS runtime was slightly worse on these particular examples, due to overhead involved in maintaining the various data structures in our prototype implementation. The fact that our approach allows equality checks to be skipped is vital in other domains where the checker/verifier may be much slower than NFA bisimulation.

Q3: How does ReGiS compare with existing regex optimizers?

Although existing regex optimizers may terminate quickly, they do not guarantee minimality of the result regular expressions. We show that our regular expression optimization tool produces high-quality results compared to existing tools, thereby validating design choices such as our regular expression cost metric. We identified several existing open-source regex optimization tools, *Regexp-Optimizer* [62] (we will refer to this as *Opt03*), *RegexOpt*, [27] (we will refer to this as *RegOpt*), and *Regular Expression Gym*, a part of the *Noam* project [63], and installed them locally on our workstation. We needed a set of "ground truth" regular expressions for which we know the minimum-cost equivalent expressions. Thus, we selected the regular expressions from Q1 for which our tool reported a minimum, giving us 2176 inputs, and from these we selected only those containing at least one Kleene star, giving us 1574. We then transformed each of these into an equivalent expression known to be superlinear—we used the RXXR regular expression static analysis tool [26, 45] to check whether an expression was vulnerable, and if not, we randomly applied semantics-preserving transformations known to increase complexity, e.g., $\mathsf{a}^* \longrightarrow \mathsf{a}^*\mathsf{a}^*$. This was repeated until RXXR confirmed vulnerability. Finally, RXXR provided us with an attack string, allowing us to target the vulnerability.

Using these superlinear expressions, we ran ReGiS and the other optimizers. The output expression from each was given to the PCRE regex engine, and matched against the respective attack string. We found that the number of PCRE steps is generally proportional to PCRE runtime. Ultimately, the number of steps provides a more "implementation-independent" measure of regular expression complexity than runtime, since it roughly corresponds to number of steps in the NFA traversal (Section 4.4). Figure 9 shows the number of PCRE steps on the respective regular expressions. Most points appear above the diagonal, meaning ReGiS produced lower-cost regular expressions in terms of PCRE matching complexity.

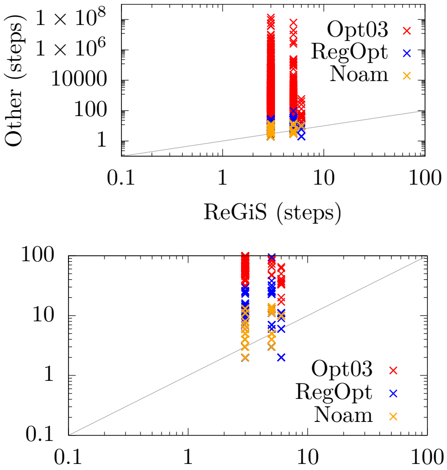

Figure 9: Quality of ReGiS results versus open-source regex optimizers: (a) Number of PCRE steps to match against attack string; (b) Detail near diagonal.

6 DISCUSSION AND FUTURE WORK

While our benchmarks are small, they are exhaustive in the sense that *all expressions* having the given structure/bounds are included. Tools like RXXR [26, 45] accept an input regex and identify a *vulnerable subexpression* leading to superlinear behavior, and this subexpression is often smaller than the input, meaning that handling small expressions has real-world value. For example, in a 2793-regex dataset that the RXXR authors mined from the internet, RXXR identifies 122 regexes as vulnerable—in 99 of these, the vulnerable subexpression has length 50 or less, even though the vulnerable input regexes have lengths up to 1067 (vulnerable subexpressions were up to 53x smaller than the inputs, averaging 3.9x smaller).

Even focusing on small exhaustive regular expression benchmarks, serious problems can arise (Figure 2 shows superlinear behavior of a single-character regular expression), and we have shown that existing optimizers fail to offer workable solutions.

While we believe ReGiS to be an important step toward the high-level goal of scaling up synthesis, especially in regards to regular expression optimization, there are engineering and research challenges we plan to address in future work.

- Our regular expression optimization technique handles "pure" regular expressions, while many real-world regexes go beyond this core language. It is not fundamentally difficult to extend support, but this will require giving Egg more flexible rewrite-rule functionality, such as operations on character classes. Note that the primary ingredient of superlinear behavior is alternation under Kleene star, leading to exponential backtracking (Section 4.4)—no functionality outside pure regular expressions is needed to trigger this.
- Solver frameworks powered by DPLL rely on *heuristics* to improve search performance. There are similar opportunities for carefully-designed heuristics here. For example, we

assign Unifiers based on lowest overlay edge cost, but these could also be chosen based on "similarity" of contained expressions, increasing likelihood of fast equality checks.

- Additional optimizations are possible. *Verification* is often a bottleneck in synthesis—for us, this is an NFA equality check, which is usually fast, but we could, e.g., parallelize several equality checks within each pair of E-classes, utilizing the fastest result. *Incremental* node/expression cost maintenance in the E-graph would also improve performance. We found that the Hopcroft-Karp (HK) NFA bisimulation algorithm which exploits equivalence classes does not seem to offer improvement over a naïve on-the-fly NFA-to-DFA bisimulation check on the DAG-like Thompson NFAs. We plan to investigate other equality-checking techniques [6].

7 RELATED WORK

Program Synthesis and Superoptimization

Jeon et al. [25] tackle the synthesis scalability problem by handling multiple enumeration steps in parallel. Alur et al. [3] use a divide-and-conquer approach, partitioning the set of inputs, solving a smaller synthesis problem within each partition, and then combining the results together. Superoptimization [47, 44] is an approach for optimizing sequences of instructions. In contrast, we perform rewriting (syntactic) and enumeration (semantic) steps in parallel—the interplay between these is key to our approach.

Combining Rewriting with Synthesis

Huang et al. [24] describe an approach which combines parallelism with a divide-and conquer methodology to perform synthesis with enumeration and deduction (conceptually similar to rewriting). ReGiS offers additional parallelization opportunities, by allowing the enumeration and rewriting to happen in parallel.

Using a rewriting-based approach for expression optimization requires a technique for overcoming *local minima* in the rewriting. We achieve this by combining equality saturation-based rewriting with syntax-guided synthesis. Nandi et al. [38] leverage equality saturation [55, 59] while performing search for CAD model decompilation, but they do not offer global minimality guarantees, or cost functions beyond expression size. They overcome local minima by speculatively adding non-semantics-preserving rewrites, and "undo" these later, after the final expression has been extracted.

Cosy [30, 31] enumeratively synthesizes data structures, using a *lossy* "deduplication" mechanism to maintain equivalence classes. Our approach compactly maintains *all* equivalence class members, not just representatives. Smith et al. [51] combine synthesis with rewriting, but require a *term-rewriting system* (TRS). Constructing a TRS is undecidable, so human input (using a proof assistant) is typically needed, while our approach is fully-automated.

There are machine learning-based approaches that combine synthesis-like search with rewriting [49, 15]. The key distinction between these approaches and ours is the need for *training data*, whereas our approach is designed to operate without this. These data-driven approaches typically cannot guarantee *global* optimality, and also cannot propose *new* rules—said another way, they offer a purely syntactic approach to optimization.

A related topic is *synthesizing rewrite rules*, which has been investigated in the context of security hardware/software [29] and SMT [40]. Another related direction is *theory exploration*, which uses E-graphs to enumerate lemmas for theorem proving [50].

Regular Expression Denial of Service (ReDoS) Attacks

Algorithmic complexity attacks are well-known, with early research in this area focusing on network intrusion detection and other systems-related functions [17, 52]. Catastrophic backtracking and attacks against the complexity of regular expressions have also begun to appear in the literature [19], but while some useful rule-of-thumb guides have been available for some time [22, 54, 46], *general awareness* of these vulnerabilities may not be widespread.

Vulnerable Regular Expression Detection

Existing work *detects* regular expressions vulnerable to ReDoS. *Static analysis* can find the complexity of a regular expression, e.g., Berglund et al. [4] formalize regular expression matching in Java, and statically determine whether a given Java regular expression has exponential runtime. Weideman et al. [58] build on that work, providing more precise characterization of worst-case runtime.

A related problem is finding an *attack string* that causes poor performance on a given regular expression. ReScue [48] does this via genetic search and properties of the *pumping lemma*. RXXR [26, 45] finds an attack string, and a vulnerable *subexpression* causing superlinear behavior on that string. Rexploiter [60] constructs an *attack automaton*, characterizing the entire language of attack strings. These approaches are complimentary to ours, which seeks to *remove vulnerabilities* from known-vulnerable regular expressions.

ReDoS Attack Prevention

Some authors have suggested updating existing backtracking algorithms with a *state cache* [18], which *fully* memoizes traversal of the NFA, achieving polynomial runtime at the expense of significant memory usage. Since developers may seek more power than core regular expressions provide, other work seeks to extend the expressibility of Thompson-like approaches to support additional features like backreferences [37].

Synthesis from examples [14, 42] is related to our work. This requires input-output examples, and an expression is automatically constructed to fit these examples. The work does not directly address ReDoS, but could potentially be leveraged for that purpose.

8 CONCLUSION

We present rewrite-guided synthesis (ReGiS), a new approach for expression optimization that interfaces syntax-guided synthesis (SyGuS) with equality saturation-based rewriting. We leverage ReGiS to address the problem of optimizing superlinear regular expressions, demonstrating the power and flexibility of our framework.

ACKNOWLEDGEMENTS

We are grateful to the anonymous reviewers for their detailed comments. This work is supported by the National Science Foundation under Grant No. 2018910 and Grant No. 2124010.

REFERENCES

[1] Marco Almeida, Nelma Moreira, and Rogério Reis. "Testing the Equivalence of Regular Languages". In *J. Autom. Lang. Comb.* (2010).

[2] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. "Syn-tax-Guided Synthesis". In *Dependable Software Systems Eng.* Vol. 40. 2015, pp. 1–25.

[3] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. "Scaling Enumerative Program Synthesis via Divide and Conquer". In *TACAS.* 2017.

[4] Martin Berglund, Frank Drewes, and Brink van der Merwe. "Analyzing Catastrophic Backtracking Behavior in Practical Regular Expression Matching". In *AFL.* EPTCS. 2014.

[5] Gérard Berry and Gérard Boudol. "The Chemical Abstract Machine". In *Theor. Comput. Sci.* (1992).

[6] Filippo Bonchi and Damien Pous. "Checking NFA equivalence with bisimulations up to congruence". In *POPL.* 2013.

[7] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. "Optimizing synthesis with metasketches". In *POPL.* ACM, 2016, pp. 775–788.

[8] Frederick P. Brooks. "No Silver Bullet - Essence and Accidents of Software Engineering". In *Computer* (1987).

[9] Robert D. Cameron, Thomas C. Shermer, Arrvindh Shriraman, Kenneth S. Herdy, Dan Lin, Benjamin R. Hull, and Meng Lin. "Bitwise data parallelism in regular expression matching". In *PACT.* 2014.

[10] Cezar Câmpeanu and Kai Salomaa. "Nondeterministic Tree Width of Regular Languages". In *DCFS.* 2015.

[11] Luca Cardelli, Milan Ceska, Martin Fränzle, Marta Z. Kwiatkowska, Luca Laurenti, Nicola Paoletti, and Max Whitby. "Syntax-Guided Optimal Synthesis for Chemical Reaction Networks". In *CAV.* 2017.

[12] Pavol Cerný, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. "Efficient Synthesis for Concurrency by Semantics-Preserving Transformations". In *CAV.* 2013.

[13] Sarah Chasins and Julie L. Newcomb. "Using SyGuS to Synthesize Reactive Motion Plans". In *SYNT@CAV.* 2016.

[14] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. "Multi-modal synthesis of regular expressions". In *PLDI.* 2020.

[15] Xinyun Chen and Yuandong Tian. "Learning to Perform Local Rewriting for Combinatorial Optimization". In *NeurIPS.* Ed. by Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett. 2019.

[16] Russ Cox. *Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...)* https://swtch.com/~rsc/regexp/regexp1.html.

[17] Scott A. Crosby and Dan S. Wallach. "Denial of Service via Algorithmic Complexity Attacks". In *USENIX Security Symposium.* 2003.

[18] James C. Davis. "Rethinking Regex engines to address ReDoS". In *ESEC/SIGSOFT FSE.* 2019.

[19] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. "The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale". In *FSE.* 2018.

[20] Chen Fu, Yuxin Deng, David N. Jansen, and Lijun Zhang. "On Equivalence Checking of Nondeterministic Finite Automata". In *SETTA.* 2017.

[21] Vaibhav Gogte, Aasheesh Kolli, Michael J. Cafarella, Loris D'Antoni, and Thomas F. Wenisch. "HARE: Hardware accelerator for regular expressions". In *MICRO.* 2016.

[22] Jan Goyvaerts. *Runaway Regular Expressions: Catastrophic Backtracking.* https://www.regular-expressions.info/catastrophic.html.

[23] Juraj Hromkovic, Sebastian Seibert, Juhani Karhumäki, Hartmut Klauck, and Georg Schnitger. "Communication Complexity Method for Measuring Nondeterminism in Finite Automata". In *Information and Computation* (2002).

[24] Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. "Reconciling enumerative and deductive program synthesis". In *PLDI.* 2020.

[25] Jinseong Jeon, Xiaokang Qiu, Armando Solar-Lezama, and Jeffrey Foster. "Adaptive Concretization for Parallel Program Synthesis". In *CAV.* 2015.

[26] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. "Static Analysis for Regular Expression Denial-of-Service Attacks". In *NSS.* 2013.

[27] Dan Kogai. *Regexp::Optimizer.* https://metacpan.org/pod/release/DANKOGAI/Regexp-Optimizer-0.15/lib/Regexp/Optimizer.pm.

[28] Dexter Kozen. "A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events". In *LICS.* 1991.

[29] DongKwon Lee, Woosuk Lee, Hakjoo Oh, and Kwangkeun Yi. "Optimizing homomorphic evaluation circuits by program synthesis and term rewriting". In *PLDI.* 2020.

[30] Calvin Loncaric, Michael D. Ernst, and Emina Torlak. "Generalized data structure synthesis". In *ICSE.* 2018.

[31] Calvin Loncaric, Emina Torlak, and Michael D. Ernst. "Fast synthesis of fast collections". In *PLDI.* 2016.

[32] Jan van Lunteren, Christoph Hagleitner, Timothy Heil, Giora Biran, Uzi Shvadron, and Kubilay Atasu. "Designing a Programmable Wire-Speed Regular-Expression Matching Accelerator". In *MICRO.* 2012.

[33] Konstantinos Mamouras, Kaiyuan Yang, Lingkun Kong, Qixuan Yu, Agnishom Chattopadhyay, Alexis Le Glaunec, and Yi Huang. "Software-Hardware Codesign for Efficient In-Memory Regular Pattern Matching". In *PLDI.* 2022.

[34] Richard Mayr and Lorenzo Clemente. "Advanced automata minimization". In *POPL.* 2013.

[35] Jedidiah McClurg, Hossein Hojjat, and Pavol Cerný. "Synchronization Synthesis for Network Programs". In *CAV.* 2017.

[36] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In *TACAS.* Lecture Notes in Computer Science. 2008.

[37] Kedar S. Namjoshi and Girija J. Narlikar. "Robust and Fast Pattern Matching for Intrusion Detection". In *INFOCOM.* 2010.

[38] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. "Synthesizing structured CAD models with equality saturation and inverse transformations". In *PLDI.* 2020.

[39] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. "Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)". In *J. ACM* 53.6 (2006), pp. 937–977.

[40] Andres Nötzli, Andrew Reynolds, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Clark W. Barrett, and Cesare Tinelli. "Syntax-Guided Rewrite Rule Enumeration for SMT Solvers". In *SAT.* 2019.

[41] Alexandros Palioudakis, Kai Salomaa, and Selim G Akl. "Quantifying nondeterminism in finite automata". In *Annals of the U. of Bucharest* (2015).

[42] Rong Pan, Qinheping Hu, Gaowei Xu, and Loris D'Antoni. "Automatic Repair of Regular Expressions". In OOPSLA (2019).

[43] Daniele Parravicini, Davide Conficconi, Emanuele Del Sozzo, Christian Pilato, and Marco D. Santambrogio. "CICERO: A Domain-Specific Architecture for Efficient Regular Expression Matching". In *ACM Trans. Embed. Comput. Syst.* (2021).

[44] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodík, and Dinakar Dhurjati. "Scaling up Superoptimization". In *ASPLOS.* 2016.

[45] Asiri Rathnayake and Hayo Thielecke. "Static Analysis for Regular Expression Exponential Runtime via Substructural Logics". In *CoRR* abs/1405.7058 (2014).

[46] Alex Roichman and Adar Weidman. *Regular Expression Denial of Service.* https://www.checkmarx.com/wp-content/uploads/2015/03/ReDoS-Attacks.pdf.

[47] Eric Schkufza, Rahul Sharma, and Alex Aiken. "Stochastic superoptimization". In *ASPLOS.* 2013.

[48] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. "ReScue: crafting regular expression DoS attacks". In *ASE.* 2018.

[49] Rohit Singh and Armando Solar-Lezama. "SWAPPER: A framework for automatic generation of formula simplifiers based on conditional rewrite rules". In *FMCAD.* 2016.

[50] Eytan Singher and Shachar Itzhaky. "Theory Exploration Powered By Deductive Synthesis". In *CoRR* abs/2009.04826 (2020).

[51] Calvin Smith and Aws Albarghouthi. "Program Synthesis with Equivalence Reduction". In *VMCAI.* 2019.

[52] Randy Smith, Cristian Estan, and Somesh Jha. "Backtracking Algorithmic Complexity Attacks against a NIDS". In *ACSAC.* 2006.

[53] Cristian-Alexandru Staicu and Michael Pradel. "Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers". In *USENIX Security Symposium.* USENIX Association, 2018, pp. 361–376.

[54] Bryan Sullivan. *Regular Expression Denial of Service Attacks and Defenses.* https://docs.microsoft.com/en-us/archive/msdn-magazine/2010/may/security-briefs-regular-expression-denial-of-service-attacks-and-defenses.

[55] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. "Equality Saturation: A New Approach to Optimization". In *LMCS* (2011).

[56] Ken Thompson. "Regular Expression Search Algorithm". In *Commun. ACM* 11.6 (1968), pp. 419–422.

[57] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. "TRANSIT: specifying protocols with concolic snippets". In *PLDI.* ACM, 2013, pp. 287–296.

[58] Nicolaas Weideman, Brink van der Merwe, Martin Berglund, and Bruce W. Watson. "Analyzing Matching Time Behavior of Backtracking Regular Expression Matchers by Using Ambiguity of NFA". In *CIAA.* LNCS. 2016.

[59] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. "egg: Fast and extensible equality saturation". In *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–29.

[60] Valentin Wüstholz, Oswaldo Olivo, Marijn J. H. Heule, and Isil Dillig. "Static Detection of DoS Vulnerabilities in Programs that Use Regular Expressions". In *TACAS.* 2017.

[61] Yi-Hua E. Yang and Viktor K. Prasanna. "Optimizing Regular Expression Matching with SR-NFA on Multi-Core Systems". In *PACT.* 2011.

[62] Joel Yliluoma. *Perl-compatible regular expression optimizer.* https://bisqwit.iki.fi/source/regexopt.html.

[63] Ivan Zuzak and Vedrana Jankovic. *Regular Expression Gym (Noam).* http://ivanzuzak.info/noam/webapps/regex_simplifier/.

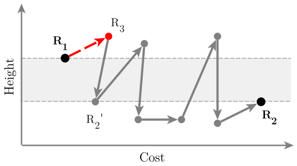

Figure 10: Lemma A.1 base case.

A PROOFS OF THEOREMS

Theorem 3.1 (Soundness). *If ReGiS returns an expression e' for an optimization instance $(e, cost, W, [\![\cdot]\!], \approx)$, then e' is no larger than the minimal expression in $hl(e)$, i.e., $cost(e') \leq cost(e'')$ for any $e'' \in hl(e)$.*

Proof. If the algorithm returns an expression e', then the Figure 4 machine's final step must have been either Saturate or Union2 (these are the only rules that allow termination).

In the case of Saturate (with a *complete* rewrite rule set W), by definition of E-graph saturation, all possible equalities (modulo the rewrite rules W) have been incorporated into the E-graph E, meaning the source E-class $class(E, e)$ contains all grammatically valid expressions that are syntactically equivalent to e. Let e_m be a minimal expression in $hl(e)$. By completeness of W, since $[\![e_m]\!] \approx [\![e]\!]$, we know that $e_m \in class(E, e)$. Thus, since the return value is $E_{min}(S_0) = E_{min}(e) = min(class(E, e))$, we know that $cost(E_{min}(S_0)) \leq cost(e_m)$.

In the case of Union2, we know that S_0 and the minimal unprocessed target T_k are equivalent, since they are in the same E-class. No target T_j (where $j < k$) is equal to the source, since the index k can only be incremented by Union3 when processing an inequality. Because Enumerate adds all elements of $hl(e)$ to T in increasing order of cost, we have that T_k is the lowest-cost element of $hl(e)$ that is equivalent to the source. Thus, $E_{min}(S_0)$ is no greater than the cost of the minimal element of $hl(e)$. □

Theorem 3.2 (Completeness). *If e_m is a minimal expression in $hl(e)$, i.e., $e_m \in hl(e)$ and $cost(e_m) \leq cost(e'')$ for any $e'' \in hl(e)$, then ReGiS's result e' will have $cost(e') = cost(e_m)$.*

Proof. Enumerate adds every expression with height no greater than $height(e)$ to T in increasing order of cost. Thus, the minimal element $e_m \in hl(e)$ will be added to T at some index j, and $T_i \not\approx e$ for all $i < j$. Because new overlay graph edges initially have unit weight, once e_m appears as T_j, Equality is able to register the equality $e_m = e$, and Union1 will be able to place e_m and e into the same E-class. For each target T_i where $i < j$, Inequality and Union3 will ensure that the index k of the minimal unprocessed target T_k is incremented, resulting in $k = j$. At this point, the Union2 rule will allow the algorithm to terminate with $E_{min}(S_0)$, which has cost no greater than e_m. □

Lemma A.1. *Consider regular expressions R_1, R_2. If $L(R_1) = L(R_2)$, $height(R_1) > height(R_2)$, $|\{R \mid L(R_1) = L(R) = L(R_2)$ and $height(R_1) > height(R) > height(R_2)\}| = 0$, and $cost(R_1) \leq cost(R_2)$, then $\exists R'$ such that $L(R') = L(R_2)$, $height(R') \leq height(R_2)$, and $cost(R') \leq cost(R_1)$.*

Proof. Figure 10 shows this graphically. There are no regular expressions in the vertical space between R_1 and R_2. The path connecting R_1, R_2 is formed by a sequence of rewrite rules. No segment (application of a single rewrite rule) can angle down and to the right or up and to the left, since no individual rewrite rule increases cost while reducing height (or vice versa).

We proceed by induction over length n of the path from R_1 to R_2. Based on the previously-mentioned segment angle constraint, the smallest possible n is 2.

Base case $n = 2$. This case is visualized by the path $R_1 \longrightarrow R_3 \longrightarrow R_2'$ in Figure 10. If R_3 is below the shaded area, it cannot be to the right of R_1 due to the angle constraint, meaning $cost(R_3) \leq cost(R_1)$, and we are finished with the proof, since we have found an $R' = R_3$ such that $cost(R') \leq cost(R_1)$ and $height(R') \leq height(R_2')$.

Consider the case where R_3 is above the shaded area. In this case, the edge from R_3 to R_2' represents a decrease in height. Examining only *height-reducing* rewrite rules used in our algorithm (Section 4.3), we are limited to the following possibilities for R_2', R_3 (where A is any regular expression).

rule	R_2'	R_3	R_1
3	A	$A + 0$	$A + 0, \boxed{A} + 0, 0 + A, A$
4	A	$A + A$	$A + A, \boxed{A} + A, A + \boxed{A}, A$
6	A	$1 \cdot A$	$1 \cdot A, 1 \cdot \boxed{A}, A$
7	A	$A \cdot 1$	$A \cdot 1, \boxed{A} \cdot 1, A$
10	0	$0 \cdot A$	$0 \cdot A, 0 \cdot \boxed{A}, 0$
11	0	$A \cdot 0$	$A \cdot 0, \boxed{A} \cdot 0, 0$
16	A^*	$1 + A{\cdot}A^*$	$1 + A{\cdot}A^*, 1 + \boxed{A}{\cdot}A^*, 1 + A{\cdot}\boxed{A^*},$ $1 + A{\cdot}\boxed{A}^*, A{\cdot}A^* + 1, A^*$
17	A^*	$1 + A^*{\cdot}A$	$1 + A^*{\cdot}A, 1 + A^*{\cdot}\boxed{A}, 1 + \boxed{A^*}{\cdot}A,$ $1 + \boxed{A}^*{\cdot}A, A^*{\cdot}A + 1, A^*$
18	A^*	$A^*{\cdot}A^*$	$A^*{\cdot}A^*, \boxed{A^*}{\cdot}A^*, A^*{\cdot}\boxed{A^*}, \boxed{A}^*{\cdot}A^*,$ $A^*{\cdot}\boxed{A}^*, A^*$
19	A^*	A^{**}	$A^{**}, \boxed{A^*}^*, \boxed{A}^{**}, A^*$
20	1	1^*	$1^*, 1$
21	1	0^*	$0^*, 1$

Based on these options for R_3, the corresponding options for R_1 are listed, using $\boxed{R}$ to denote a single rewrite rule applied to some subexpression of R, such that $height(\boxed{R}) \leq height(R)$. In all of these cases for R_1, we can find an R' such that $cost(R') \leq cost(R_1)$ and $height(R') \leq height(R_2')$. In any case that does not contain $\boxed{A^*}$, we can simply let $R' = R_2'$. For cases that contain $\boxed{A^*}$, if A is of the form R^* for some R, then we can let $R' = A$. Otherwise, the $\boxed{A^*}$ can instead be written as $\boxed{A}^*$, meaning we can again let $R' = R_2'$.

Inductive step $n > 2$. Assume the property holds for all $k < n$. Given the path of length n between R_1 and R_2, consider the first segment $R_1 \longrightarrow R_3$ (shown as the red/dashed arrow in Figure 10). If R_3 is *below* the shaded area, it cannot be to the right of R_1 due to the angle constraint, meaning $cost(R_3) \leq cost(R_1)$ and $height(R_3) \leq height(R_2)$, so we are finished with the proof (we have found $R' = R_3$).

Otherwise, if R_3 is *above* the shaded area, we apply the induction hypothesis to R_3, R_2, giving us R_4 such that $cost(R_4) \leq cost(R_3)$ and $height(R_4) \leq height(R_2)$. If R_4 is to the left of R_1, we are finished ($R' = R_4$). Otherwise, we can apply the induction hypothesis to R_1, R_4, giving us R_5 such that $cost(R_5) \leq cost(R_1)$ and $height(R_5) \leq height(R_4)$. Since $height(R_4) \leq height(R_2)$, we have $height(R_5) \leq height(R_2)$, and we are finished (we found $R' = R_5$). □

Theorem 4.1 (Height vs. Cost). *Consider regular expressions R_1, R_2. If $L(R_1) = L(R_2)$, $height(R_1) > height(R_2)$, and $cost(R_1) \leq cost(R_2)$, then $\exists R'$ such that $L(R') = L(R_2)$, $height(R') \leq height(R_2)$, and $cost(R') \leq cost(R_1)$.*

Proof. We proceed by induction over the number of regular expressions with height between that of R_1 and R_2, i.e., $n = |\{R \mid height(R_1) > height(R) > height(R_2)\}|$.

Base case $n = 0$. This follows from Lemma A.1.

Inductive step $n > 0$. Assume the Theorem holds for all $k < n$. Assume $height(R_1) > height(R_2)$ and $cost(R_1) \leq cost(R_2)$. Let R_3 be a regular expression such that $height(R_1) > height(R_3) > height(R_2)$.

Case I: $cost(R_3) > cost(R_1)$. Applying the induction hypothesis to R_1, R_3, we can obtain an R_4 such that $height(R_4) \leq height(R_3)$ and $cost(R_4) \leq cost(R_1)$. If $height(R_4) \leq height(R_2)$, we are finished with the proof (we found $R' = R_4$). Otherwise if $height(R_4) > height(R_2)$, since we have $cost(R_4) \leq cost(R_2)$ from the induction hypothesis, we can apply the induction hypothesis to R_4, R_2 to obtain an R_5 such that $height(R_5) \leq height(R_2)$ and $cost(R_5) \leq cost(R_4)$. Since $cost(R_4) \leq cost(R_1)$, we are finished with the proof ($R' = R_5$).

Case II: $cost(R_3) \leq cost(R_1)$. Using the induction hypothesis, we have $cost(R_3) \leq cost(R_2)$. Applying the induction hypothesis to R_3, R_2, we can obtain an R_5 such that $height(R_5) \leq height(R_2)$ and $cost(R_5) \leq cost(R_3)$. By Case II assumption, we have $cost(R_5) \leq cost(R_1)$, so we are finished with the proof ($R' = R_5$). □