SWAPPER: A Framework for Automatic Generation of Formula Simplifiers based on Conditional Rewrite Rules

Rohit Singh and Armando Solar-Lezama
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139
Email: {rohitsingh,asolar}@csail.mit.edu

***Abstract*—This paper addresses the problem of creating simplifiers for logic formulas based on conditional term rewriting. In particular, the paper focuses on a program synthesis application where formula simplifications have been shown to have a significant impact. We show that by combining machine learning techniques with constraint-based synthesis, it is possible to synthesize a formula simplifier fully automatically from a corpus of representative problems, making it possible to create formula simplifiers tailored to specific problem domains. We demonstrate the benefits of our approach for synthesis benchmarks from the SyGuS competition and automated grading.**

I. INTRODUCTION

Formula simplification plays a key role in SMT solvers and solver-based tools. SMT solvers, for example, often use local rewrite rules to reduce the size and complexity of the problem before it is solved through some combination of abstraction refinement and theory reasoning [1], [2]. Moreover, many applications that rely on solvers often implement their own formula simplification layer to rewrite formulas before passing them to the solver [3], [4], [5].

One important motivation for tools to implement their own simplification layer is that formulas generated by a particular tool often exhibit patterns that can be exploited by a custom formula simplifier but which would not be worthwhile to exploit in a general solver. Unfortunately, writing a simplifier by hand is challenging, not only because it is difficult to come up with the simplifications and their efficient implementation, but also because some simplifications can actually make a problem harder to solve by the underlying solver. This means that producing a simplifier that actually improves solver performance often requires significant empirical analysis.

In this paper, we present SWAPPER, a framework for automatically generating a formula simplifier from a corpus of benchmark problems. The input to SWAPPER is a corpus of formulas from problems in a particular domain. Given this corpus, SWAPPER generates a formula simplifier tailored to the common recurring patterns in this corpus and empirically tuned to ensure that it actually improves solver performance.

SWAPPER operates in four phases. In the first phase (1), the system uses representative sampling to identify common repeating sub-terms in the different formulas in the corpus. In the second phase (2), these repeating sub-terms are passed to the rule synthesizer which generates conditional simplifications that can be applied to these sub-terms when certain local conditions are satisfied. These conditional simplifications are the simplification rules which in the third phase (3) must be compiled to actual C++ code that will implement these simplifications. In the fourth phase (4), SWAPPER uses auto-tuning to evaluate combinations of rules based on their empirical performance on a subset of the corpus (Training set) in an effort to identify a subset of the rules that maximizes solver performance.

In this paper, we focus SWAPPER on formulas generated by the SKETCH synthesis solver [6]. We choose SKETCH solver because it is already very efficient, and it has been shown to be faster than the most popular SMT solvers on the formulas that arise from encoding synthesis problems [7]. A major part of this performance comes from carefully tuned formula rewriting, so improving the performance of this solver is an ambitious target. The SKETCH synthesizer has been applied to a number of distinct domains which include storyboard programming [8], query extraction [9], automated grading (Autograder) [10], sketching Java programs [11], SyGus competition benchmarks [12], synthesizing optimal CNF encodings [13] and programming of line-rate switches [14]. Crucially for our purposes, we have available to us large numbers of benchmark problems that are clearly identified as coming from some of these different domains. For example, SKETCH has over a thousand benchmarks for Autograder problems obtained from student submissions to introductory programming assignments on the edX platform. For this paper, we will focus on two important domains: Autograder and SyGus competition benchmarks and present a small case study with the CNF (SAT) encodings benchmarks.

This paper makes the following key contributions:

1) We demonstrate how to automate the process of generating conditional rewrite rules specific to the common recurring patterns in formulas from a given domain.
2) We demonstrate the use of autotuning to select an optimal subset of rules and generate an efficient simplifier.
3) We evaluate our approach on multiple domains from the SKETCH synthesizer and show that the generated simplifiers reduce the synthesis times of SKETCH by 15%-60% relative to the existing SKETCH with its hand-crafted simplifier.

II. OVERVIEW OF SWAPPER

SWAPPER takes as input a corpus of problems (formulas) $\{P\} \subset \mathcal{D}$ that are assumed to be from the same domain $\mathcal{D}$ and therefore share certain structural features—what it means for problems to come from a given domain is left ambiguous,

but the assumption underlying our work is that problems that come from the same domain will have similar structure and will respond similarly to simplifications. Given $\{P\}$, the goal of SWAPPER is to produce a formula rewriter such that for any $P \in \mathcal{D}$ from the domain, it rewrites P to $P' = rewrite(P)$, where P' is logically equivalent to P but easier to solve.

The rewriters produced by SWAPPER are term rewriters that work by making local substitutions when a known pattern is encountered in a context satisfying certain constraints. For example, the rewrite rule below indicates that when the guard predicate (*pred*) $b < d$ is satisfied, one can locally substitute the pattern on the left hand side (LHS): $or(lt(a,b), lt(a,d))$ with the smaller pattern on the right hand side (RHS).

$$or(lt(a,b), lt(a,d)) \xrightarrow{b<d} lt(a,d)$$

SWAPPER's approach is to automatically generate large collections of such rules and then compile them to an efficient rewriter for the entire formula. Making this approach work requires addressing four key challenges: (1) Choosing promising LHS patterns (2) Finding the best rewrite rule for a given LHS (3) Generating an efficient implementation that applies these rules and (4) Making sure that the rules actually improve the performance of the solver. In the rest of this section, we outline how each of the components of SWAPPER address these challenges.

A. Pattern Finding

The first phase of SWAPPER addresses the problem of identifying promising LHS patterns for rewrite rules. In order to do this, the pattern finder takes as input a corpus of formulas $\{P\}$ and uses a representative sampling scheme (explained in Sec. III) to find frequently recurring patterns in the formulas from the corpus. The idea is that rewrite rules that target patterns that occur frequently in the corpus are more likely to have a high impact in the complexity of the overall formula.

In addition to identifying high frequency patterns, this phase also identifies properties that tend to hold when those patterns occur. Specifically, we assume that the rewriter has access to a function $static(a)$ that for any sub-term a of a larger formula P, can determine an over-approximation of the range of values that a can take when the free variables in P are assigned values from their respective ranges. In the rewriter that is currently part of the SKETCH solver, for example, this function is implemented by performing abstract interpretation over the formula.

For a given pattern, the pattern finder keeps a set of contexts, where each context corresponds to the information captured by $static$ on an occurrence of that pattern in the corpus. For example, in the case of the rewrite rule above, this phase may discover that the pattern $or(lt(a,b), lt(a,d))$ is very common, so finding a simplification rule for this pattern would be advantageous. Pattern finding may also discover that this pattern often occurs in a context where a rewriter can prove that $static(b) = (-\infty, 0]$ and $static(d) = (0, \infty)$. Using such context information, SWAPPER can learn rules that make more aggressive simplifications based on the strong assumptions.

Note that this problem is similar in essence to the Motif discovery problem [15], famous for its application in DNA fingerprinting [16]. However, existing techniques used for solving Motif discovery problem are not directly usable in SWAPPER because they lead to loss of information required for the next phase in SWAPPER (See Sec. VIII for more details).

B. Rule Synthesis

Once SWAPPER identifies promising LHS patterns for rules, together with properties of their free variables that can be assumed to hold in the contexts where these LHS patterns appear (e.g. $b \leq 0$ and $d > 0$ in the example above), the next challenge is to synthesize the rewrite rules.

A *conditional rewrite rule* has the form:

$$LHS(x) \xrightarrow{pred(x)} RHS(x),$$

where x is a vector of variables, LHS and RHS are expressions that include variables in x as free variables and *pred* is a guard predicate defined over the same free variables and drawn from a restricted grammar. The triple must satisfy the following constraint: $\forall x.\ pred(x) \implies (LHS(x) = RHS(x))$

The goal of rule synthesis is therefore twofold: (1) to synthesize predicates $pred(x)$ that can be expected to hold on at least one context identified by pattern finding for the LHS pattern, and (2) to synthesize for each of these candidate predicates, an optimal RHS for which the constraint above holds. At this stage, optimality is defined simply in terms of the size of the RHS, since it is difficult to predict the effect that a transformation will have on solution time. As we will see later, optimality in terms of size does not guarantee optimality in terms of solution time but at this stage in the process our goal is simply to identify potentially good rewrite rules. We formulate this as a Syntax-Guided Synthesis Problem [12] and solve it using the SKETCH synthesis tool [6]. The details of how the rules are synthesized are given in Sec. IV.

C. Code generation

The next challenge for SWAPPER is to generate efficient C++ code for pattern matching and replacement. To achieve this, SWAPPER builds upon the ideas of term rewrite systems like Stratego/XT [17] and GrGEN.NET [18], and performs some optimizations detailed in Sec. V. The role of this phase is similar to what the Alive system [19] does when generating peephole optimizers for LLVM.

One important optimization at this stage is rule generalization. For example, pattern finding may have discovered that the pattern $or(lt(plus(x,y),b), lt(a,d))$ was frequent, and the rule synthesis phase may have discovered the rewrite rule:

$$or(lt(plus(x,y),b), lt(a,d)) \xrightarrow{b<d} lt(plus(x,y),d)$$

Rule generalization, would identify that $plus(x,y)$ is unchanged by the rewrite rule, so the rule could be made more general by replacing $plus(x,y)$ in both patterns with a free variable to arrive at the rule shown earlier. SWAPPER needs to verify that the generalization preserves the correctness of the rule in order to avoid generating incorrect transformations. It is important to note that rule generalization doesn't affect

predicates because those have already been minimized during the Rule Synthesis step.

D. Autotuning

There are two motivations for autotuning the set of obtained rewrite rules: (1) there is a trade-off between the strength of the predicate and the reduction that can be achieved by a rule: rules with weak predicates are easier to match than rules with strong predicates, but rules with strong predicates can offer more aggressive simplification, and (2) the rules that give the most aggressive size reduction are not necessarily the best ones; for example, a rule may replace a very large LHS pattern with a small RHS but in doing so it may prevent other rules from being applied, resulting in a formula that is larger than the formula obtained without the rewriting.

For these reasons, writing optimal simplifiers based on rewrite rules is a challenging task even for human experts, which motivates our approach of using synthesis and empirical autotuning methods to automatically discover optimal sets of conditional rewrite rules. To this end SWAPPER uses OpenTuner [20], a machine learning based off-the-shelf autotuner and provides an optimization function that emits the actual performance of the solver. This phase is comparable to algorithm configuration [21], [22], which has been used for tuning parameters for SAT solvers [23]. But, unlike algorithm configuration, the optimization function in SWAPPER is based on choices of subsets and permutations of rewrite rules, and is provided to OpenTuner [20] as a black-box.

We now describe each of these phases in more detail.

III. PATTERN FINDING: SAMPLING BASED CLUSTERING

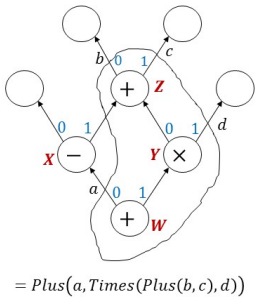
We first describe a few key features of the SKETCH synthesis system [6] which will serve both as a component and as a target for SWAPPER.

SKETCH synthesis system: SKETCH is an open source system for synthesis from partial programs. A partial program (also called a sketch) is a program with holes, where the potential code to complete the holes is drawn from a finite set of candidates, often defined as a set of expressions or a grammar. Given a partial program with a set of assertions, the SKETCH synthesizer finds a completion for the holes that satisfies the assertions for all inputs from a given input space. SKETCH uses symbolic execution to derive a predicate $P(x, c)$ that encodes the requirement that given a choice c for how to complete the program, the program should be correct under all inputs x i.e. $\exists c \forall x P(x, c)$.

SKETCH , like most solvers, represents formulas as Directed Acyclic Graphs (DAGs) in order to exploit sharing of subterms within a formula. The formulas in SKETCH involve boolean combinations of formulas involving the Theory of Arrays and Non-linear Integer Arithmetic. Because SKETCH has to solve an exists-forall ($\exists\forall$) problem, the formulas distinguish between existentially and universally quantified variables: *inputs* and *controls* respectively. The formula simplification pass that is the subject of this paper is applied to this predicate $P(x, c)$ as it is constructed and before the predicate is solved in an abstraction refinement loop based on counterexample guided inductive synthesis (CEGIS) [6].

Probabilistic Pattern Sampling: A pattern in the context of SWAPPER is an expression tree that has free variables as leafs. Our goal for the pattern finding phase is to generate a representative sample of patterns S from a corpus of DAGs $\{P\}$. In order to formalize the notion of a representative sample of patterns, we first need to define a few terms.

Consider a formula P represented as a DAG. We can define a *rooted sub-graph* of P as a sub-graph of P such that all its nodes can be reached from a selected root node in the sub-graph. A rooted sub-graph can be mapped to a pattern, where the edges at the boundary of the sub-graph correspond to the free variables in the pattern. This relationship is illustrated in Fig. 1, where we see a graph for a problem P where a rooted sub-graph has been selected, and we see the pattern that corresponds to that sub-graph. Note that a single formula P may have many sub-graphs that all correspond to the same pattern.



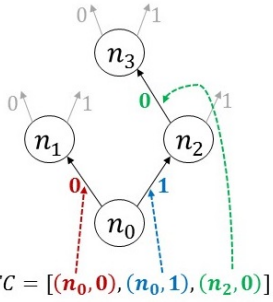


Fig. 1. Pattern from a rooted sub-graph Fig. 2. Example Tree Construction

Given a constant K, let the set $Sub_K(P)$ be the set of all rooted sub-graphs of size K of P. Given these definitions, we are now ready to state the problem of representative sampling patterns from a corpus.

Definition 1: *Representative pattern sampling* Given a corpus of problems $\{P_i\}$ and a size K, a pattern sampling approach is said to be representative if it is equivalent to sampling uniformly from the set $\bigcup_i Sub_K(P_i)$, and then mapping each of the resulting sub-graphs to their corresponding pattern.

The key problem is then how to uniformly sample the space of rooted sub-graphs in a collection of formulas. In order to describe the algorithm for this, we first build the notion of a *Tree Construction*. The formal definition is given below, but intuitively, a *Tree Construction* (TC) is a recipe for generating a tree.

Definition 2: A TC for a tree of size $K \geq 2$ and arity $\delta \geq 1$ is a list of $K-1$ pairs $[(s_i, t_i)]_{i=0}^{i=K-2}$, where each pair represents an edge that is being added to the tree. Each edge is identified by its source node s_i (which should already be in the tree) and by the index $t_i < \delta$ of the edge that is added to that source node. A TC cannot have repeated edges, so each edge adds a new node to the tree. Therefore, if n_0 is the original root node in the tree, and n_i is the node added by the i^{th} edge, then $s_i \in \{n_0, n_1, \ldots, n_i\}$ for all $i \leq K-2$.

We use *Tree* (τ) to represent the tree constructed from a TC τ in this manner.

For example, if we assume binary trees ($\delta = 2$), the following would be a valid tree construction of size 4: $\tau = [(n_0, 0), (n_0, 1), (n_2, 0)]$, and would construct the tree *Tree* (τ) as shown in Fig. 2.

Assuming trees of degree δ, it is relatively easy to uniformly sample the space of Tree Constructions for trees of size K. The idea is to keep track of the boundary of the tree (all the possible edges that have not been expanded) and to grow the tree by sampling uniformly at random from this boundary. The exact algorithm is shown below.

Algorithm 1: Uniform Sampling for Tree Constructions

input : $K \geq 2$: Size of the TC to be found
$\delta \geq 1$: Bound on number of parents of any node
$\{n_0, n_1, ..., n_{K-1}\}$: Set of K node symbols
output: τ : A Tree Construction of size K

```
τ ← List()
Λ ← List()                          ▷ maintains adjacent/boundary edges
foreach i ∈ [0, 1, ..., K − 2] do
    foreach j ∈ [0, 1, ..., δ − 1] do
        Λ.append( (n_i, j) )        ▷ adds δ edges to boundary
    (s, t) ← sample (Λ)             ▷ boundary |Λ| = ((i + 1)δ) − i
    τ.append( (s, t) )
    Λ.remove( (s, t) )              ▷ removes an edge from boundary
```

It is easy to see that any TC of size K will be sampled by Algorithm 1 with a probability of $\Pi_{0 \leq i < K-1} \frac{1}{((i+1)\delta)-i}$ because $|\Lambda| = \big((i+1)\delta\big) - i$ at the i^{th} step and we sample uniformly from Λ for each $0 \leq i < K-1$. This probability is independent of the TC being considered and hence, every TC is equally likely to be sampled by Algorithm 1.

Now, we are going to define an algorithm for representative pattern sampling that uses our ability to uniformly sample from the space of TCs (denoted by **TC**). The strategy will be as follows, given a corpus of formulas $\{P_i\}$, we are going to define a subset of the product space $\bigcup_i \text{nodes}\big(P_i\big) \times \textbf{TC}$, which we will call *Canonical*, we then define a mapping μ from *Canonical* to $\bigcup_i Sub_K(P_i)$, and we are going to show that mapping is one-to-one and onto. Finally, we will use rejection sampling [24] to sample from *Canonical* uniformly and then apply μ to in turn, sample uniformly from $\bigcup_i Sub_K(P_i)$. In the rest of the section, we define the *Canonical* set, the function μ and show that it is bijective.

Definition 3: *Canonical* set: Given a corpus of formulas $\{P_i\}$, we define *Canonical* $\subset \bigcup_i \text{nodes}\big(P_i\big) \times \textbf{TC}$ as the set of tuple-pairs $\big(\eta, \tau\big)$ with $\eta \in \text{nodes}\,(P_i)$ for some i such that:

1) It is possible to follow the TC τ at node η and construct a *rooted sub-graph* Q of P_i rooted at η i.e. there is a graph homomorphism h : *Tree* $(\tau) \mapsto P_i$ such that $h(n_0) = \eta$ and Q is the rooted sub-graph of P_i formed by the nodes corresponding to the image of h in P_i.
2) The ordering of nodes in TC τ is the same as the (unique) Breadth First Search (BFS) ordering of nodes in Q.

For example, In Fig. 1, for $(\eta, \tau) = (W, [(n_0, 1), (n_1, 0)])$ by following TC τ we obtain the homomorphism h that maps $h(n_0) = W, h(n_1) = Y, h(n_2) = Z$ and the rooted sub-graph Q corresponds to the enclosed region with nodes W, Y, Z. Also, the ordering of nodes given by τ matches the BFS ordering of nodes induced by h in Q, hence, $(\eta, \tau) \in$ *Canonical*. Note that $(W, [(n_0, 1), (n_0, 0)]) \notin$ *Canonical* because it induces the ordering W, Y, Z which is not in the BFS order W, X, Y.

Given the way we defined *Canonical*, constructing the mapping μ : *Canonical* $\mapsto \bigcup_i Sub_K(P_i)$ is straightforward: $\mu\big(\,(\eta, \tau)\,\big) = Q$ where Q is the rooted subgraph obtained by following TC τ starting at node η (Def. 3). To show that μ is onto, we consider a rooted subgraph S of P_i for some i, since any node of a rooted subgraph can be reached from the root η_S, we can construct the BFS tree BFS(S) of S and the corresponding TC τ_S that is the recipe for constructing BFS(S) so that $\mu\left((\eta_S, \tau_S)\right) = S$. To show that μ is one-to-one, we observe that for two $(\eta_1, \tau_1), (\eta_2, \tau_2)$ to map to the same rooted subgraph S, the corresponding trees should be the same (the BFS tree of S) and the ordering of nodes in τ_1, τ_2 should be the same as well (corresponding to BFS order of S), which would mean $\tau_1 = \tau_2$ and $\eta_1 = \eta_2$.

Clustering patterns: While grouping patterns together into clusters, SWAPPER considers the following: **(1) Pattern Expression**: SWAPPER builds a string of the expression represented by the pattern. The free variables in this pattern are numbered in the BFS order and the operands for commutative operations are ordered lexicographically to group together patterns when they are equivalent because of commutativity. **(2)** $static$ **function values from the benchmark formulas** : The range of values inferred by the solver (See **II-A**) for each free variable in the pattern are collected and represented as a mapping of names of the free variables to their ranges. Note that the same pattern can occur with different $static$ function values, these values are appended to one *Pattern Expression*.

Stopping Criterion: SWAPPER samples until the total number of patterns with probability of occurrence greater than a threshold ϵ converges i.e. the next M samples do not change the number of such patterns. Both ϵ and M are inputs to SWAPPER. In our experiments, we started with $M = 10,000$ and $\epsilon = 0.05$ and then increased M and decreased ϵ gradually in steps of $10,000$ and 0.01 respectively, and, sampled again until we didn't find any new patterns ($M = 50,000$ and $\epsilon = 0.02$). SWAPPER samples patterns of sizes $2, 3, 4, ...$ and stops after sampling patterns of size 7.

IV. RULE SYNTHESIS: SYNTAX-GUIDED SYNTHESIS

In this phase, SWAPPER finds corresponding rewrite rules for the set of patterns $\{Q\}$ obtained from the Pattern Finding phase. For each pattern $Q(x)$, the Pattern Finding phase also collects a set of $static$ range values (**II-A**) that can be valid for the free variables x in that pattern when it occurs as a rooted subgraph in the benchmark DAGs. We use the notation $assume(x)$ to represent a predicate over the free variables x

that evaluates to true when each free variable is in the range values given by $static$ function.

A. Problem Formulation

SWAPPER needs to find correct rewrite rules (**II-B**) for a given LHS pattern. Additionally, we want to avoid rules with predicates that will never hold in practice, so we focus on rules with predicates that are implied by the assumptions given by $static$ function obtained from the Pattern Finding Phase.

Problem 1: Given a pattern $LHS(x)$, predicate $assume(x)$ discovered by the solver for a given occurrence of the LHS pattern, and grammars for *pred*(x) and $RHS(x)$, find suitable candidates for *pred*(x) and $RHS(x)$ which satisfy the following constraints:

1) $\forall x : assume(x) \implies$ *pred*(x)
2) $\forall x : \ if$ (*pred*(x)) $then$ $(LHS(x) == RHS(x))$
3) size(RHS) < size(LHS), where size(Q) is the number of nodes in the pattern Q
4) *pred*(x) is the weakest predicate (most permissive) in the predicate grammar that satisfies previous constraints

The space of predicates: For *pred* SWAPPER employs a simple Boolean expression generator that considers conjunctions of equalities and inequalities among variables: *pred*(x) $\rightarrow$ $boolExpr(x)$ and $boolExpr(x)$ | $boolExpr(x)$, $boolExpr(x) \rightarrow x_i$ binop x_j | unop x_i where binop $\in \{<, >, =, \neq, \leq, \geq\}$, unop $\in \{\epsilon, \neg\}$.

These predicates are inspired by existing predicates present in the rules in the Sketch's hand-crafted simplifier and are easier to check statically than more complicated predicates.

Grammar for RHS: The template for RHS simulates the computation of a function using temporary variables. This computation can be naturally interpreted as a pattern. Essential grammar for the generator for RHS is shown here:

$$\begin{aligned} RHS(x) \equiv \quad & let \quad && t_1 = \textbf{simpleOp}(x); \\ & && \dots \\ & && t_k = \textbf{simpleOp}(x, t_1, ..., t_{k-1}); \\ & in\ t_k \end{aligned}$$

where **simpleOp** represents a single operation node (e.g. AND, PLUS etc) with its operands being selected from the arguments. For example, the expression $(a + b) \times c$ can be represented as: let $t_1 = Plus(a, b); t_2 = Times(t_1, c);$ in t_2. We put a strict upper bound on k as the number of nodes in the LHS for which the RHS is being searched for.

B. Correctness Constraint

Setting apart constraint 4), Problem 1 can be formulated as a classic syntax-guided synthesis [25] problem:

$$\begin{aligned} \exists c_p c_r\ \forall x \quad & \big(assume(x) \implies pred(x, c_p)\big) \\ & \wedge\ pred(x, c_p) \implies \big(LHS(x) = RHS(x, c_r)\big) \end{aligned}$$

where c_p and c_r are the choices the solver needs to make to get a concrete *pred*(x) and $RHS(x)$. Specifically, these are the choices of: (i) when to expand the grammar or when to use a terminal, and (ii) which subset of inputs to choose for a particular operation as operands. We will enforce the constraint 4) on top of solutions to the synthesis problem.

We explored two different techniques for synthesizing such rewrite rules: (1) Symbolic SKETCH based synthesis of rules, and (2) Enumerative search with heuristics. SWAPPER uses a hybrid approach to get the best of the both aforementioned techniques: scalability and exhaustiveness. We describe the hybrid technique briefly.

C. Hybrid Enumerative/SKETCH-based synthesis

SWAPPER breaks the synthesis problem into two parts:

(1) Constraints and optimizations on predicates: SWAPPER uses the enumerative approach, generates all possible candidate predicates from the specification grammar and checks for their validity based on collected assumptions $assume(x)$. It prunes the space of predicates by handling symmetries and avoiding extra work based on the result of the underlying synthesis problem (explained below). For example, if $x = (a, b)$, $assume(x) = (1 < a < 10) \wedge (b = 0)$ then some of the valid predicates will be $\{a > b, \neg b, a \geq b, a \neq b\}$.

(2) Synthesis of RHS: SWAPPER hard-codes a predicate and realizes the RHS synthesis problem in SKETCH using the **generator** and **minimize** features ([26] [27]) of the SKETCH language. In SKETCH, **generator**s are used to define the template for a grammar as a recursive program (e.g. RHS), and **minimize** keyword is used to find the smallest possible value of a computed variable in SKETCH language (e.g. size of $RHS(x)$ for a fixed *pred*(x)).

Optimization on the space of predicates: SWAPPER computes the relationship between predicates based on whether one implies the other or not for all values of the free variables e.g. $(a < b)$ implies $a \neq b$ but doesn't imply $a > b$. SWAPPER iteratively finds RHS for the least applicable predicates at any given stage. When there is no possible simplification rule for a least applicable predicate then SWAPPER can prune out all predicates implied by it because there cannot exist a rule with a more applicable predicate. This helps SWAPPER reduce overall time for the rule synthesis step.

This hybrid technique has the benefit of being able to exhaustively search for rules of big sizes while making the core synthesis problem faster (fewer constraints) and highly parallelizable (multiple SKETCH instances with different predicates are run in parallel). Note that SKETCH does synthesis of rules guaranteeing their correctness for large but bounded values of $\forall$ quantified variables (inputs), hence, we fully verify the generated rules with z3 [1] as well by expressing them as SMT constraints before using them for code generation.

V. EFFICIENT SIMPLIFIER CODE GENERATION

The code generation phase in SWAPPER implements two important optimizations. The first is *rule generalization*. As described earlier in **II-C**, the goal of rule generalization is to make the rule more applicable by eliminating redundant nodes from the LHS and RHS patterns. The second optimization is to reduce the cost of pattern matching by identifying common substructures in different patterns and avoiding redundant checks for those patterns, similar to how a compiler for a functional language would optimize pattern matching [28].

Additionally, the code generator will ensure that the pattern matching code can identify DAGs that are equivalent to a given LHS because of commutativity. Overall, the generated code is more efficient than the hand-written optimizer because the automatic code generator can optimize pattern matching without regard to the impact of optimization on the readability of the generated code.

VI. Auto-tuning Rules

SWAPPER uses OpenTuner [20] to auto-tune the set of rules according to a performance metric (based on time, memory, size of DAGs etc). OpenTuner uses an ensemble of disparate search techniques and quickly builds a model for the behavior of the optimization function treating it as a black box.

Optimization Problem Setup: SWAPPER specifies the set of all rules to the tuner and creates the following two configuration parameters: (1) A permutation parameter: for deciding the order in which the rules will be checked. (2) Total number of rules to be used.

The optimization function (**fopt**) takes as input a set of rules and returns a real number. This number corresponds to performance improvement of SKETCH on the benchmarks after rewriting them using the generated simplifier (Sec. V). The auto-tuner tries to maximize this reward by trying out various subsets and orderings of rules provided to it as input while learning a model of dependence of **fopt** on the rules.

VII. Experiments

In order to test the effectiveness of our system, we focus on three questions: (1) Can SWAPPER generate good simplifiers in reasonable amounts of time and with low cost of computational power? (2) How do the simplifiers generated by SWAPPER affect SMT solving performance of SKETCH relative to the hand written simplifier in SKETCH? (3) How domain specific are the simplifiers generated by SWAPPER?

For evaluation of SWAPPER on SKETCH domains, we compared the following three simplifiers:

1) Hand-crafted: This is the default simplifier in SKETCH that has been built over a span of eight years. It comprises of simplifications based on (a) rewrite rules that can be expressed in our framework (**II-B**) (b) constant propagation (c) structure hashing [29] and (d) a few other complex simplifications that cannot be expressed in our framework
2) Baseline: This disables the rewrite rules that can be expressed in our framework from the Hand-crafted simplifier but applies the rest of the simplifications (b)-(d).
3) Auto-generated: This incorporates the SWAPPER's auto-generated rewrite rules on top of the Baseline simplifier.

Now, we elaborate on the details of the experiments.

A. Domains and Benchmarks:

Domain	Benchmark DAGs Used	Avg. Number of Terms
AutoGrader	45	23289
Sygus	22	68366

We investigated benchmarks from two domains of SKETCH applications. Sygus corresponds to the SyGus competition benchmarks translated from SyGus format to Sketch specification [12] and AutoGrader ones are obtained from student's assignment submissions in the introduction to programming online edX course [10]. For each of these domains we picked suitable candidates for SWAPPER's application by (1) eliminating those benchmarks which did not have more than 5000 terms in the formula represented by their DAGs and those which took less than 5 seconds to solve - so that there's enough patterns and opportunity for improvement (2) removing those which took more than 5 minutes to solve - this was done to keep SWAPPER's running time reasonable because we need to run each benchmark multiple times during auto-tuning phase. Using these cutoffs, the total number of usable benchmarks for AutoGrader domain were reduced from 2404 to 45 and for Sygus from 309 to 22.

B. Synthesis Time and Costs are Realistic:

To generate a simplifier, SWAPPER employed a private cluster running Openstack as the infrastructure for parallelized computations with parallelisms of 20-40 on two virtual machines emulating 24 cores, 32GB RAM of processing power each. A worst case estimate of the cost of computation done by SWAPPER based on our experiments using the Amazon Web Services [30] estimator is presented below. SWAPPER can be used to automatically synthesize a simplifier for a very reasonable cost (less than $50)

Domain	Pattern Finding	Rule Synthesis	Auto-Tuning	Total Time (hours)	Cost
AutoGrader	3 hours	1 hour $\times 5$	0.08×150	20	$22
Sygus	2 hours	1 hour $\times 5$	0.1×150	22	$24

C. SWAPPER Performance

To test the performance of SWAPPER on SKETCH benchmarks from a particular domain, we divided the corpus into three disjoint sets randomly $(\text{SEARCH}, \text{TRAIN}, \text{TEST})$. The SEARCH set was used to find patterns in the domain and TRAIN set was used in the auto-tuning phase for evaluation. And finally, TEST set was used to empirically confirm that the generated simplifier is indeed optimal for the domain. Moreover, we used 2-fold cross validation to ensure that there was no over-fitting on TRAIN set. We achieved this by exchanging TRAIN and TEST sets and auto-tuning with the TEST set instead of the TRAIN set. We obtained similar performing simplifiers as a result and verified that there was no over-fitting.

We implemented the evaluation of benchmarks in SWAPPER as a python script that takes a set of DAGs as input, runs SKETCH on each of them multiple times (set to 5 in our experiments) and obtain the quartile values(3 points that cut data into 4 equal parts including the median) for time taken. In the graphs presenting SKETCH solving times, we show the upper and lower quartiles around the median with dotted or shaded lines of the same color as the thick line depicting the median time. Also, note that we will not consider simplification time in these experiments because of it being a one-time negligible (a fraction of a second) time-step as compared to further SKETCH solving.

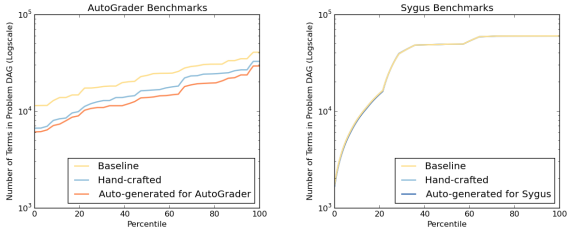


Fig. 3. Change in sizes with different simplifiers

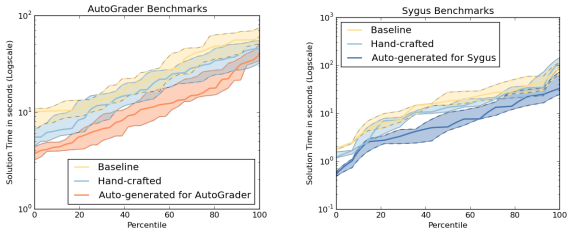


Fig. 4. Median running time percentiles with Quartile confidence intervals

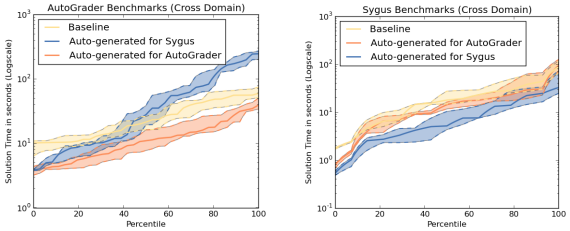


Fig. 5. Domain specificity of Auto-generated simplifiers: Time distribution

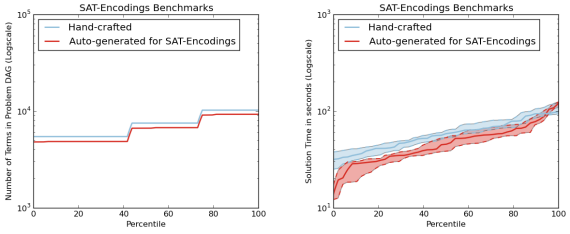


Fig. 6. SAT-Encodings domain case study

We obtained 301 rules for AutoGrader domain and 105 rules for Sygus domain. The optimal simplifier for AutoGrader used 135 of the rules and the one for Sygus used 65 rules.

Benefits over the existing simplifier in SKETCH: Auto-generated simplifier reduced the size of the problem DAGs by 13.8% (AutoGrader) and 1.1%(Sygus) on average as compared to the size of DAGs obtained after running Hand-crafted simplifier (Figure 3). On DAGs obtained after using Auto-generated simplifier on average, SKETCH solver performed better than on those obtained by using Hand-crafted simplifier (Figure 4): (1) Auto-generated simplifier made SKETCH run faster on 80% of the AutoGrader benchmarks and 90% of the Sygus benchmarks (2) The average times taken by SKETCH to solve a benchmark simplified using Auto-generated simplifier were 13s (AutoGrader) and 8s (Sygus) as compared to 20s and 21s respectively for the Hand-crafted simplifier. Figures 3 and 4 show distribution of sizes and times for SKETCH solving after applying all three simplifiers with percentiles on the x-axis. It clearly shows the consistent improvement in performance by applying the Auto-generated simplifier. Note that Sygus benchmarks are written at a level of abstraction that is very close to the DAGs in Sketch and hence there aren't many opportunities for size reduction for these problems.

We found that there are two reasons why the auto-generated rules improved upon the hand-crafted rules: (1) The synthesizer discovered rules with large LHS patterns that were not present in the hand-crafted optimizer. (2) The autotuner was able to discover that some rules caused a performance degradation even when they reduced the formula size.

Benefits over the unoptimized version of SKETCH: Auto-generated simplifier reduced the size of the problem DAGs by 38.6% (AutoGrader) and 1.6%(Sygus) on average (Figure 3). Application of Auto-generated simplifier results into huge improvements in running times for SKETCH solver on both AutoGrader and Sygus benchmarks as compared to application of Baseline: the average time of solving a benchmark was reduced from 27.5s (AutoGrader) and 22s (Sygus) to 13s and 8s respectively (Figure 4).

Domain specificity: We took the Auto-generated simplifier obtained from one domain and used it to simplify benchmarks from the other domain and then ran SKETCH on the simplified benchmarks. Application of Auto-generated simplifier obtained from Sygus increased the SKETCH running times drastically on a few AutoGrader benchmarks when compared to the application of Baseline simplifier (Figure 5), and, resulted into SKETCH running slower than after application of Auto-generated simplifier obtained from AutoGrader domain. Application of Auto-generated simplifier generated for AutoGrader domain reduced the running times of SKETCH solver on average as compared to the Baseline on Sygus benchmarks but the times were still far away from the performance gains obtained by application of Auto-generated simplifier generated for the Sygus domain (Figure 5). This validates our hypothesis of these generated simplifiers to be very domain specific.

D. SAT Encodings Domain We performed an additional case study using SWAPPER on problems generated during synthesizing optimal CNF (SAT) encodings [13]. We used a subset of 70 benchmarks with solution times between 30s and 100s and divided it randomly into SEARCH, TRAIN, TEST sets with $21, 22, 27$ benchmarks respectively. We compare the Hand-crafted simplifier against the Auto-generated simplifier for this domain in Fig. 6. SWAPPER generated 117 rules and, on average, the SKETCH solving time reduced from 58.8s to 51.1s, the DAG sizes were reduced by 11%.

VIII. RELATED WORK

A pre-processing step in constraint solvers and solver-based tools (like Z3, Boolector [2], SKETCH etc) is an essential one and term rewriting has been extensively used as a part this pre-processing step [31], [4], [5], [3]. These pre-processing steps are very important and can have a significant impact on performance.

Each part of our framework solves an independent problem and is different from the state of the art, specialized for our purposes. A recent paper introducing Alive [19], a domain specific language for specifying, verifying, and compiling peephole optimizations in LLVM is the closest to our framework as a whole. Their rewrite rules are guarded by a predicate, they use static analyses to find the validity of those guards, they verify the rules and then compile them to efficient C++ code for rewriting LLVM code: all similar to our phases. However, their system is targeted towards the compilers community and relies upon the developers to discover and specify rewrite rules. Our work is targeted towards the solver community and automatically synthesizes the rewrite rules from benchmark problems of a given domain.

In the context of Motif discovery problem [15] (finding recurrent sub-graphs) recently we have seen some attempts to use machine learning [32] and distributed algorithms [33] to compute the Motifs as quickly as possible. Our DAGs, on the other hand, have labeled nodes and our motifs have to account for symmetries due to commutative nodes, which makes direct translation to Motif discovery problem more difficult.

In the superoptimization community, people explore all possible equivalent programs and find the most optimal one. One could view SWAPPER as a superoptimizer for formula simplifiers. Superoptimizing an individual formula will be too expensive, but [34] came up with the idea of packaging the superoptimization into multiple rewrite rules similar to what we are doing here except in the context of programs. Although it looks similar in spirit to our work, there are a few differences. Most importantly, [34] uses enumeration of potential candidates for optimized instruction sequences and then checks if it is indeed most optimal. Whereas, we use a hybrid approach that primarily relies on constraint based synthesis for generating the rules, which offers a possibility of specifying a structured grammar for the functions.

The third phase in SWAPPER automatically generates simplifier's code is similar to a term or graph rewrite system like Stratego/XT [17] or GrGEN.NET [18]. They offer declarative languages for graph modeling, pattern matching, and rewriting. Both the tools generate efficient code for program/graph transformation based on rule control logic provided by the user. We build upon their ideas and develop our own compiler because we already had an existing framework for simplification (SKETCHSimplifier). Our strategy is comparable with LALR parser generation [35] where the next look-ahead symbol helps decide which rule to use.

Acknowledgment: We thank the DARPA MUSE grant FA8750-14-2-0242 for supporting this project.

REFERENCES

[1] L. De Moura and N. Bjørner, "Z3: an efficient smt solver," ser. TACAS'08/ETAPS'08. Springer-Verlag, 2008, pp. 337–340.

[2] R. Brummayer and A. Biere, "Boolector: An efficient smt solver for bit-vectors and arrays," ser. TACAS '09, 2009, pp. 174–177.

[3] S. Chandra, S. J. Fink, and M. Sridharan, "Snugglebug: a powerful approach to weakest preconditions," in *PLDI '09*, 2009, pp. 363–374.

[4] A. Cheung, A. Solar-Lezama, and S. Madden, "Partial replay of long-running applications," ser. ESEC/FSE '11. ACM, 2011, pp. 135–145.

[5] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *USENIX Conference OSDI '08*.

[6] A. Solar-Lezama, "The sketching approach to program synthesis," in *APLAS*, 2009, pp. 4–13.

[7] J. P. Inala, X. Qiu, B. Lerner, and A. Solar-Lezama, "Type assisted synthesis of recursive transformers on algebraic data types," *CoRR*, vol. abs/1507.05527, 2015.

[8] R. Singh and A. Solar-Lezama, "Synthesizing data structure manipulations from storyboards," 2011.

[9] A. Cheung, A. Solar-Lezama, and S. Madden, "Inferring sql queries using program synthesis," *CoRR*, vol. abs/1208.2013, 2012.

[10] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," *SIGPLAN Not.*, vol. 48, no. 6, pp. 15–26, Jun. 2013.

[11] J. Jeon, X. Qiu, J. S. Foster, and A. Solar-Lezama, "Jsketch: sketching for java," E. D. Nitto, M. Harman, and P. Heymans, Eds. ACM, 2015.

[12] R. Alur, D. Fisman, R. Singh, and A. Solar-Lezama, "Results and analysis of sygus-comp'15," 2015, pp. 3–26.

[13] J. P. Inala, R. Singh, and A. Solar-Lezama, "Synthesis of domain specific CNF encoders for bit-vector solvers," in *SAT 2016*, 2016.

[14] A. Sivaraman, M. Budiu, A. Cheung, C. Kim, S. Licking, G. Varghese, H. Balakrishnan, M. Alizadeh, and N. McKeown, "Packet transactions: A programming model for data-plane algorithms at hardware speed," *CoRR*, vol. abs/1512.05023, 2015.

[15] G. K. K. Sandve and F. Drabløs, "A survey of motif discovery methods in an integrated framework." *Biology direct*, Apr. 2006.

[16] N. C. Jones and P. A. Pevzner, *An Introduction to Bioinformatics Algorithms (CMB)*. MIT Press, Aug. 2004.

[17] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser, "Stratego/XT 0.17. A language and toolset for program transformation," *Science of Computer Programming*, vol. 72, no. 1-2, pp. 52–70, 2008.

[18] R. Geiß, G. V. Batz, D. Grund, S. Hack, and A. M. Szalkowski, "GrGen: A Fast SPO-Based Graph Rewriting Tool," pp. 383 – 397, 2006.

[19] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr, "Provably correct peephole optimizations with alive," *SIGPLAN Not.*, vol. 50, no. 6, pp. 22–32, Jun. 2015.

[20] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U. O'Reilly, and S. P. Amarasinghe, "Opentuner: an extensible framework for program autotuning," in *PACT '14*.

[21] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "Paramils: An automatic algorithm configuration framework," *J. Artif. Int. Res.*, 2009.

[22] C. Ansótegui, M. Sellmann, and K. Tierney, "A gender-based genetic algorithm for the automatic configuration of algorithms," in *CP '09*.

[23] F. Hutter, M. T. Lindauer, A. Balint, S. Bayless, H. H. Hoos, and K. Leyton-Brown, "The configurable SAT solver challenge (CSSC)," *CoRR*, vol. abs/1505.01221, 2015.

[24] A. Leon-Garcia, *Probability, Statistics, and Random Processes for Electrical Engineering*, 3rd ed. Pearson/Prentice Hall, 2008.

[25] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, A. Udupa *et al.*, "Syntax-guided synthesis." IEEE, 2013, pp. 1–8.

[26] A. Solar-Lezama, "Program synthesis by sketching," Ph.D. dissertation, Berkeley, CA, USA, 2008, aAI3353225.

[27] R. Singh, R. Singh, Z. Xu, R. Krosnick, and A. Solar-Lezama, "Modular synthesis of sketches using models," in *VMCAI 2014*.

[28] M. Pettersson, "A term pattern-match compiler inspired by finite automata theory," in *CC' 92*. Springer-Verlag, 1992, pp. 258–270.

[29] A. Mishchenko, S. Chatterjee, and R. Brayton, "Dag-aware aig rewriting a fresh look at combinational logic synthesis." ACM, 2006.

[30] "Amazon Web Services," Online. [Online]. Available: http://aws.amazon.com/

[31] N. B. Leonardo de Moura, "Smt: Techniques, hurdles, applications," *SAT/SMT Summer School, MIT*, 2011. [Online]. Available: http://research.microsoft.com/en-us/um/people/leonardo/mit2011.pdf

[32] M. A. Kon, Y. Fan, D. Holloway, and C. DeLisi, "Svmotif: A machine learning motif algorithm," ser. ICMLA '07, pp. 573–580.

[33] Y. Liu, B. Schmidt, and D. L. Maskell, "An ultrafast scalable many-core motif discovery algorithm for multiple gpus," ser. IPDPSW '11.

[34] S. Bansal and A. Aiken, "Automatic generation of peephole superoptimizers," ser. ASPLOS XII. ACM, 2006, pp. 394–403.

[35] R. N. Horspool, "Incremental generation of lr parsers," *Computer languages*, vol. 15, pp. 205–233, 1989.