Automatically Learning Optimal Formula Simplifiers and Database Entity Matching Rules

by

Rohit Singh

Bachelor of Technology (Honors), Computer Science and Engineering, Indian Institute of Technology Bombay (2011)
Master of Science, Electrical Engineering and Computer Science, Massachusetts Institute of Technology (2013)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctorate of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2017

Author .....
Department of Electrical Engineering and Computer Science
August 31, 2017

Certified by .....
Armando Solar-Lezama
Associate Professor
Department of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Professor Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students
Department of Electrical Engineering and Computer Science

Automatically Learning Optimal Formula Simplifiers and Database Entity Matching Rules

by

Rohit Singh

Submitted to the Department of Electrical Engineering and Computer Science
on August 31, 2017, in partial fulfillment of the
requirements for the degree of
Doctorate of Philosophy

Abstract

Traditionally, machine learning (ML) is used to find a function from data to optimize a numerical score. On the other hand, synthesis is traditionally used to find a function (or a program) that can be derived from a grammar and satisfies a logical specification. The boundary between ML and synthesis has been blurred by some recent work [56,90]. However, this interaction between ML and synthesis has not been fully explored. In this thesis, we focus on the problem of finding a function given large amounts of data such that the function satisfies a logical specification and also optimizes a numerical score over the input data. We present a framework to solve this problem in two impactful application domains: formula simplification in constraint solvers and database entity matching (EM).

First, we present a system called Swapper based on our framework that can automatically generate code for efficient formula simplifiers specialized to a class of problems. Formula simplification is an important part of modern constraint solvers, and writing efficient simplifiers has largely been an arduous manual task. Evaluation of Swapper on multiple applications of the Sketch constraint solver showed 15-60% improvement over the existing hand-crafted simplifier in Sketch.

Second, we present a system called EM-Synth based on our framework that generates as effective and more interpretable EM rules than the state-of-the-art techniques. Database entity matching is a critical part of data integration and cleaning, and it usually involves learning rules or classifiers from labeled examples. Evaluation of EM-Synth on multiple real-world datasets against other interpretable (shallow decision trees, SIFI [116]) and non-interpretable (SVM, deep decision trees) methods showed that EM-Synth generates more concise and interpretable rules without sacrificing too much accuracy.

Thesis Supervisor: Armando Solar-Lezama
Title: Associate Professor
Department of Electrical Engineering and Computer Science

Acknowledgments

I would first like to thank my adviser Armando Solar-Lezama. Armando has always given me quality advice and motivation for every research project we have worked on together. Without his guidance and mentorship, I wouldn't have been able to grow as a researcher, a presenter and work on such impactful projects. I would like to thank my collaborator and PhD committee member Samuel (Sam) Madden. I really value all the insights and feedback I received from Sam while working on the entity-matching project and while writing this thesis. I would like to thank my RQE and PhD committee member Adam Chlipala. I appreciate all of Adam's feedback that has helped me ask and answer important questions about my research projects and their presentation in this thesis.

I would also like to thank my undergraduate adviser Supratik Chakraborty, my internship advisers Thomas Henzinger, Krishnendu Chatterjee and Barbara Jobstmann, who introduced me to the world of formal verification and synthesis. The research projects that I worked on as an undergraduate with these researchers got me excited about the possibilities of using formal methods for automation and motivated me to pursue a PhD in program synthesis.

Research collaborators: I am thankful to Nan Tang, Vamsi Meduri, Ahmed Elmagarmid, Paolo Papotti and Jorge-Arnulfo Quiane-Ruiz for their guidance and constant effort to make the entity matching project successful. I would also like to thank Jeevana Inala, Jack Feser, Zhilei Xu and Shachar Itzhaky for their hard work and persistence while working on our respective collaborative projects.

CAP, MIT-PL and CSAIL: I am grateful for being a part of the CAP research group and the larger MIT-PL/CSAIL community. There was always someone whom I could talk to about anything related to PL or computer science research, or anything interesting in general. I really value the research discussions I've have and the time spent with all the CAP members. Thank you Evan, Ivan, Jack, Jean, Jeevana, Jimmy, Kevin, Kuat, Nadia, Rishabh, Shachar, Xiaokang, Zenna, Zhilei. I've learned a lot from everyone in the this community and I'm looking forward to building upon that knowledge in the future.

Family: Finally, I would like to thank the most important part of my life – my family. I wouldn't have been here without the support of my siblings and my parents. First and foremost, I want to thank my mother for her dedication and sacrifice. She raised me and my

siblings alone after my father fell victim to an untimely disease. There is no amount of words that can express my gratitude towards her. I want to thank my dad for everything he did for me and the family. I hope to continue contributing to his legacy. My brother Rishabh, who was also a PhD student in the CAP group, has always been a role model and a father figure for me. I am grateful for having him by my side in every big decision of my life including the research projects that have shaped this thesis. My sister Richa, has always been the most mature and caring of us all siblings. Thank you Richa di for helping me become a responsible and caring person, and emulate those characteristics in my research style. I also want to thank my sister-in-law Deeti di and brother-in-law Rajul for their constant support and encouragement.

I dedicate this thesis to my mother, who has selflessly done everything she can to help her kids succeed in life without asking for anything in return. It is only because of her I am here writing this thesis.

Contents

1 Introduction **15**

1.1 Formula simplification with Swapper 21

1.2 Database entity matching with EM-Synth 24

1.3 Key Contributions 28

1.3.1 Automatic generation of formula simplifiers 29

1.3.2 Synthesis of concise EM rules 29

1.4 Thesis Overview 31

2 Synthesis of components in application domains **33**

2.1 Syntax-guided synthesis (SyGuS) framework 34

2.2 Synthesis of conditional rewrite rules in Swapper 35

2.2.1 Formula simplification in Sketch 36

2.2.2 Core rule-synthesis problem 37

2.2.3 Space of expressions and predicates 40

2.2.4 Hybrid enumerative/symbolic synthesis in Swapper 42

2.3 Synthesis of EM rules in EM-Synth 44

2.3.1 Notation and **EM-GBF** rule-synthesis problem 44

2.3.2 Core SyGus Formulation 47

2.3.3 Numerical search for EM thresholds in Sketch 49

3 Specialization information extraction **53**

3.1 Specialization information in Swapper 53

3.2 Representative sampling of patterns in Swapper 55

3.3 Specialization information in EM-Synth 61

3.4 Choosing sets of examples in EM-Synth 62

3.4.1 Synthesis from a few EM examples (CEGIS) . . . 63
3.4.2 Synthesis with inconsistent examples (RANSAC) . . . 65

4 Assembly of components **67**
4.1 Assembly in SWAPPER . . . 68
4.1.1 Soundness of assembly . . . 69
4.1.2 Generalization of rewrite rules . . . 71
4.1.3 LALR-style pattern matching in SWAPPER . . . 76
4.2 Assembly in EM-Synth . . . 77
4.2.1 Boolean combinations of EM rules . . . 77
4.2.2 Consensus of EM rules . . . 79

5 Best assembly tuning **81**
5.1 Combinatorial auto-tuning in SWAPPER . . . 81
5.2 Tuning in EM-Synth . . . 83
5.2.1 **EM-GBF** optimization problem . . . 83
5.2.2 Tuning algorithms in EM-Synth . . . 84

6 Shared framework infrastructure **89**

7 SWAPPER system evaluation **93**
7.1 System Design & Implementation . . . 93
7.2 Experiments . . . 97
7.2.1 Domains and Benchmarks . . . 97
7.2.2 Synthesis Time and Costs are Realistic . . . 98
7.2.3 SWAPPER Performance . . . 99
7.2.4 SAT Encodings Domain . . . 102
7.2.5 Analysis of Generated Rules and their Impact . . . 102

8 EM-Synth system evaluation **105**
8.1 Algorithms and optimizations in EM-Synth . . . 105
8.1.1 Incremental grammar bounds in RS-CEGIS . . . 105
8.1.2 Sampling: bias in picking examples in RS-CEGIS . . . 106
8.1.3 Algorithms for entity matching using EM-Synth . . . 106

8.1.4 Bucketing-based optimized EM-rule testing . . . 107
8.2 System design and implementation . . . 108
8.2.1 Feature processing . . . 109
8.2.2 EM algorithms . . . 111
8.2.3 Experiment infrastructure . . . 112
8.3 Experimental setup . . . 113
8.3.1 Datasets . . . 113
8.3.2 Performance and interpretability metrics used . . . 115
8.3.3 Similarity functions used . . . 115
8.3.4 Input features for ML techniques . . . 115
8.3.5 Comparisons with state-of-the-Art ML approaches . . . 115
8.3.6 Comparisons with rule-based learning approaches . . . 116
8.3.7 Techniques and parameters . . . 117
8.3.8 Performance evaluation . . . 118
8.4 Experimental results . . . 118
8.4.1 Exp-1: Interpretability . . . 119
8.4.2 Exp-2: Effectiveness vs. interpretable decision trees . . . 124
8.4.3 Exp-3: Effectiveness vs. expert-provided rules . . . 125
8.4.4 Exp-4: Effectiveness vs. non-interpretable methods . . . 126
8.4.5 Exp-5: Variable training data . . . 126
8.4.6 Exp-6: Efficiency of training . . . 129
8.4.7 Exp-7: Efficiency of Testing . . . 130
8.4.8 Exp-8: Impact of the custom synthesizer in SKETCH . . . 133

9 Related Work **135**
9.1 Overall framework . . . 135
9.1.1 Combining program synthesis and machine learning . . . 135
9.1.2 Program synthesis with quantitative objectives . . . 136
9.1.3 Synthesis of components . . . 137
9.2 SWAPPER system . . . 137
9.2.1 Formula rewriting in constrain solvers . . . 138
9.2.2 Pattern finding . . . 138

9.2.3 Comparison with superoptimization . . . 138
9.2.4 Code generation . . . 139
9.3 EM-Synth system . . . 139
9.3.1 Machine Learning-Based Entity Matching . . . 139
9.3.2 Rule-based entity matching . . . 139
9.3.3 Active learning and crowdsourcing . . . 140
9.3.4 Program synthesis for databases . . . 140
9.3.5 Special-purpose constraint solvers . . . 141

10 Conclusion **143**

A Synthesis with Sketch: implementation notes **145**
A.1 Sketch synthesis system . . . 145
A.2 Swapper Sketch formulation . . . 147
A.3 Sketch formulation for **EM-GBF** rule synthesis . . . 150

List of Figures

1-1 Overall synthesis/learning framework 21
1-2 Overall phases for the SWAPPER system 23
1-3 Instantiation of the overall framework (Fig. 1-1) as SWAPPER 24
1-4 Sample tables for persons 26
1-5 High-level description for EM-Synth framework 27
1-6 Instantiation of the overall framework (Fig. 1-1) for EM-Synth 28

2-1 An example grammar of a simple Boolean expression 34
2-2 Relationship between inputs $(x_1, x_2, \ldots, x_n)$ of a pattern Q and the corresponding sub-terms $(a_1, a_2, \ldots, a_n)$ of the formula P 37
2-3 The language of formula expressions in SKETCH. 40

3-1 Pattern from a rooted sub-graph 56
3-2 Example tree construction 57

4-1 An example truth table for $\mathcal{C}_\varphi = \{\varphi_1, \varphi_2, \varphi_3\}$ 78

7-1 SWAPPER implementation and experiments overview 94
7-2 Change in sizes with different simplifiers 100
7-3 Median running-time percentiles with quartile confidence intervals 101
7-4 Domain specificity of the Auto-generated simplifiers: Time distribution 101
7-5 SAT-Encodings domain case study 102
7-6 An example rule generated for AutoGrader benchmarks 103

8-1 Some examples of similarity functions with their corresponding hashing-function families 108
8-2 EM-Synth implementation and experiments overview 109

8-3 Dataset statistics . . . 114
8-4 Input-similarity functions ($\mathcal{F}$) . . . 116
8-5 Interpretability results for 5-folds experiment (80% training and 20% testing data) . . . 120
8-6 User interpretability preference: Cora, Amazon-GoogleProducts (AGP), Locu-FourSquare (LFS), DBLP-Scholar (DBLP) . . . 123
8-7 Effectiveness results for 5-folds experiment (80% training and 20% testing data) 124
8-8 Interpretability user study form for the participants . . . 125
8-9 Effectiveness results for 5-folds experiment: RS-Consensus vs. non-interpretable methods . . . 126
8-10 Locu-Foursquare (100 runs with 99% CIs on the means in the shaded regions) 127
8-11 Cora (100 runs with 99% CIs on the means in the shaded regions) . . . 127
8-12 Amazon-GoogleProducts (100 runs with 99% CIs on the means in the shaded regions) . . . 128
8-13 DBLP-Scholar (100 runs with 99% CIs on the means in the shaded regions) . . . 128
8-14 Efficiency of training (average time for training per fold) for 5-folds experiment (80% training / 20% testing) . . . 129
8-15 Efficiency of testing a classifier: SVM vs. RuleSynth-generated rule on all pairs and bucketed pairs . . . 130
8-16 Efficiency of testing a classifier: RuleSynth-generated rule on all pairs vs. bucketed pairs . . . 131
8-17 Efficiency of testing . . . 131
8-18 Time taken by traditional Sketch, Z3 [41] solver and Sketch with custom synthesizer (Sketch Optimized). . . . 132

List of Algorithms

1 Hybrid enumerative/symbolic SKETCH based approach 43

2 Custom synthesizer for EM-Synth inside SKETCH 50

3 Uniform sampling for TCs 58

4 Probabilistic Sampling for Pattern Finding 60

5 CEGIS-based specialization-information extraction (RS-CEGIS) 64

6 RS-RANSAC algorithm for specialization information extraction in EM-Synth 66

- Procedure Incremental grammar bounds 106

Chapter 1

Introduction

Traditionally, a machine learning (ML) problem consists of finding a function f given some data D as input, such that a numerical score $Score(f, D)$ is optimized. The data may be provided in a supervised (input and output examples) or unsupervised (only inputs) fashion. A synthesis problem, on the other hand, traditionally consists of finding a function f that satisfies a logical specification on all inputs to f. This requires logical reasoning about f and the specification in a formal theory (e.g., theory of linear integer arithmetic) and constraining the function f to have a logical structure as well (e.g., f can be constrained to be a program in a restricted programming language). The boundary between ML and synthesis has been blurred by some recent work. For example, programming by example (PBE) [56] applies techniques from the synthesis community to synthesize programs from small amounts of data. Unlike ML problems, the data is treated as a logical specification, but there is also a numerical score that quantifies how well the program will generalize to other data. Another recent work [90] presents a framework for supervised learning of programs from large amounts of input-output examples with some noise. However, there is an entire space of ways in which ML and synthesis can interact which has not been fully explored.

In this thesis, we focus on the problem of finding a function f given large amounts of data D such that it satisfies a logical specification and also optimizes a numerical score over the same data. In other words, we would like to get the best of both worlds (traditional ML and synthesis). Unlike PBE and the work from [90], we target a general problem that takes large amounts of data as input, provided in a supervised or unsupervised fashion.

However, we do assume that the synthesis problem has some inherent logical structure such that the function f can be generated from a set of *components*. Unlike traditional synthesis of functions from components [91], we need to synthesize each component from the data.

Many practical applications adhere to these assumptions e.g., formula simplification [106] and SAT encoding [62] in constraint solvers, data wrangling with programming by example (PBE) tools [56, 95, 99] and machine learning (ML) problems that also require finding a program as the model [90, 104]. In this thesis, we will discuss two of these applications, namely: (1) formula simplification in constraint solvers and (2) database entity matching from examples. We start by presenting our framework in Fig. 1-1 and discussing the formalism below.

Formal discussion

A traditional machine learning problem consists of finding a function $f^* : A \rightarrow B$ given some data D and a numerical score function $Score(f, D)$ such that:

$$f^* = \underset{f:A \rightarrow B}{\text{argmax}} \; Score\,(f, D)$$

where $D \subseteq A$ (unsupervised) or $D \subseteq A \times B$ (supervised).

A traditional synthesis problem consists of finding a function (or program) $f^* : A \rightarrow B$ with some logical structure usually described by a grammar G with symbols over the vocabulary of a formal theory $\mathcal{T}$ such that a logical specification in the same theory $\mathcal{T}$ is satisfied i.e.,

$$f^* \in G \wedge \left(\forall_{in \in A} \; Spec\,(f^*, in)\right)$$

Programming by example (PBE) [56] is a sub-field of synthesis where the logical specification comes from small amounts of data $d = \{(in, out) : in \in A, out \in B\}$ provided as a set of input-output examples and f^* is ranked based on a pre-computed function $rank$:

$$f^* = \underset{f:A \rightarrow B \bigwedge \Phi(f)}{\text{argmax}} \; rank\,(f)$$
$$\text{where } \Phi\,(f) \equiv \Big(f \in G \wedge \forall_{(in,out) \in d} \; \big(f(in) = out\big)\Big)$$

Problem 1 *We focus on the problem of finding a function $f^* : A \rightarrow B$ given data D (in supervised or unsupervised fashion), a grammar G, a logical specification $Spec$ and a numerical score $Score$ such that:*

$$f^* = \underset{f:A\rightarrow B \bigwedge \Phi(f)}{\operatorname{argmax}} \; Score\,(f, D)$$
$$where\ \Phi\,(f) \equiv \Big(f \in G \wedge \ \forall_{in \in A}\ Spec\,(f, in) \Big)$$

i.e., we have a logical specification over the function f while optimizing a numerical score computed over f and D.

In general, this problem is computationally very hard because reasoning about both the numerical score and the logical specification together can lead to complicated search spaces, especially when the numerical score is itself complicated and not amenable to symbolic reasoning. So, in this thesis, we focus on a special case of this problem. Instead of synthesizing the complicated function f directly, we synthesize simpler *components* of f and assume that there is a way to *assemble* the components together to build a correct f. Note that unlike traditional synthesis from components from a predefined library [91], we need to first synthesize our components before assembling them. More formally, we assume that:

1. The function f can be constructed from some **components** $C_1, C_2, \ldots, C_n$ (for some $n \geqslant 1$) with each $C_i : A' \rightarrow B'$ (for $1 \leqslant i \leqslant n$) being from a grammar G_C: i.e., there is a procedure $Assemble$ that can generate f from its components:

   $$f = Assemble(\pi, \mathcal{C}), \text{ where } \mathcal{C} = \{C_1, C_2, \ldots, C_n\}$$

   and $\pi \in \Pi$ is a parameter such that each value of π corresponds to a different way of combining the components in $\mathcal{C}$. Note that the number (n) of components that $Assemble$ takes as input does not have to be fixed.

2. There are logical specifications $Spec_{C_i}$ ($1 \leqslant i \leqslant n$) for the components C_i such that satisfaction of these specifications for each component implies that f satisfies its

specification $Spec$ as well i.e.,

$$\forall_{\pi \in \Pi} \Bigg(\Big(f = Assemble(\pi, \mathcal{C}) \wedge \bigwedge_{i=1}^{n} \big(C_i \in G_C \wedge Spec_{C_i}(C_i) \big) \Big) \implies f \in G \wedge \Big(\forall_{in \in A}\ Spec(f, in) \Big) \Bigg), \text{ where } \mathcal{C} = \{C_1, C_2, \ldots, C_n\}$$

As mentioned before, the motivation behind tackling this special case is rooted in efficiency. In this special case, we can focus on smaller synthesis problems of synthesizing the components and solve them efficiently. Moreover, we do not have to verify correctness of f (which may be expensive) because of the property (2.).

Note that property (2.) is valid for all values of $\pi \in \Pi$. This enables us to explore the space Π of the parameter π to optimize the numerical $Score$. In other words, different choices of π may lead to different numerical scores, but they will all ensure that the logical specification $Spec$ is satisfied by f. And, we can search in this space to find an optimal value of π that optimizes the numerical $Score$.

Additionally, the $Spec_{C_i}$ specification for a component $C_i : A' \rightarrow B'$ can encode more constraints than just the correctness of the component on all inputs to it. The motivation behind adding these extra constraints is to synthesize components that are not only correct but also likely to lead to a high value of the numerical $Score$. Formally, we can rewrite the $Spec_{C_i}$ specification as:

$$Spec_{C_i}(C_i) \equiv \Big(\Big(\forall_{in' \in A'}\ Spec_{correct}\left(C_i, in'\right) \Big) \wedge Spec_{score}(C_i) \Big) \tag{1.1}$$

It is important to mention here that we do not guarantee the global optimality of the score (similar to how traditional ML algorithms do not guarantee global optimality), but we will produce a local optimum instead in the space of functions given by the restricted choices of the components and the space of parameters in the assembly of these components. Problem 2 summarizes the special case of Problem 1 that we address in this thesis.

Problem 2 *In this thesis, we focus on a special case of Problem 1 i.e., finding a function $f^* : A \rightarrow B$ given data D (in supervised or unsupervised fashion), a grammar G, a logical specification Spec and a numerical score Score such that*

$$f^* = \underset{f : A \rightarrow B \bigwedge \Phi'(f)}{\operatorname{argmax}} \ Score\,(f, D)$$

where $\Phi'(f)$ specifies the following:

1. *There is a grammar G_C and* ***components*** *$C_1, C_2, \ldots, C_n$ ($n \geqslant 1$) with each $C_i : A' \rightarrow B'$ (for $1 \leqslant i \leqslant n$) being from the grammar G_C and there is a procedure Assemble parametrized by $\pi \in \Pi$ to generate f from these components:*

$$f = Assemble(\pi, \mathcal{C})$$
$$where\ \mathcal{C} = \{C_1, C_2, \ldots, C_n\}, \pi \in \Pi$$

2. *There are logical specifications $Spec_i$ ($1 \leqslant i \leqslant n$) for the components C_i, respectively, such that logical satisfaction of $Spec_i$ for each component C_i implies that f satisfies its specification Spec i.e.,*

$$\forall_{\pi \in \Pi} \Bigg(\Big(f = Assemble(\pi, \mathcal{C}) \wedge \bigwedge_{i=1}^{n} \left(C_i \in G_C \wedge Spec_{C_i}\left(C_i\right) \right) \Big) \implies f \in G \wedge \Big(\forall_{in \in A}\ Spec\left(f, in\right) \Big) \Bigg),\ where\ \mathcal{C} = \{C_1, C_2, \ldots, C_n\}$$

Now we present a framework to solve Problem 2.

Framework details

Our framework follows the strategy of synthesizing multiple components $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$ and then finding the optimal parameter π for the assembly of these components. To synthesize a component C_i, we construct $Spec_{score}$ introduced in Equation (1.1) by extracting *specialization information* from the data and then generating the synthesis specification from it.

We present the framework to solve Problem 2 in Fig. 1-1. The framework consists of four steps:

1. **Specialization information extraction**: in this step, *specialization information* α_i is extracted from the data D. From this α_i we generate the specification $Spec_{C_i}$ i.e.,

$$Spec_{C_i}(C_i) \equiv \Big(\Big(\forall_{in' \in A'}\ Spec_{correct}\left(C_i, in'\right) \Big) \wedge Spec_{score}(\alpha_i, C_i) \Big)$$

2. **Synthesis of components**: in this step, a component C_i is synthesized from the specification $Spec_{C_i}$. The synthesis problem corresponds to solving:

$$\exists_{C_i} \left(C_i \in G_C \wedge Spec_{C_i}(C_i) \right)$$

This step may send some feedback (for example, the generated component C_i itself) to the first step. This feedback or some other heuristic may be used to decide the next specialization information α_i to be selected by the first step. This iterative process will have a domain-specific end condition as to when this loop should stop. At the end of this iterative process, there is a collection of synthesized components $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$ as the output of this step.

3. **Assembling components**: this step uses the *Assemble* procedure to build a candidate function f. To begin with, it may choose arbitrary values for the parameters in π, but afterwards the choice of these parameters is guided by the next step.

4. **Best assembly tuning**: this step uses the candidate function f and data D to evaluate its score $Score(f, D)$ and provides feedback to the previous step for selecting the next set of parameters π. The **assembling components** step generates the next candidate function f and sends it back to this step in an iterative manner. This iterative process will end based on a domain-specific end condition and output an optimal f that corresponds to the best score seen so far.

Framework application domains

This framework has been applied to three different application domains: (1) formula simplification in constraint solvers [106], (2) database entity matching [104], and (3) CNF encoding in SMT solvers [62]. We consider the first two domains in this thesis. In general, this framework can also be applied to other synthesis problems that require optimizing a complicated numerical score (e.g., programming by example [56], synthesis of bitstream programs [90]).

Depending on the application domain, each step has a specialized interpretation and processes that are responsible for performing the action of the corresponding step in that domain. We present the two application domains considered in this thesis and discuss the corresponding instantiations of this overall framework in Sec. 1.1 (formula simplification)

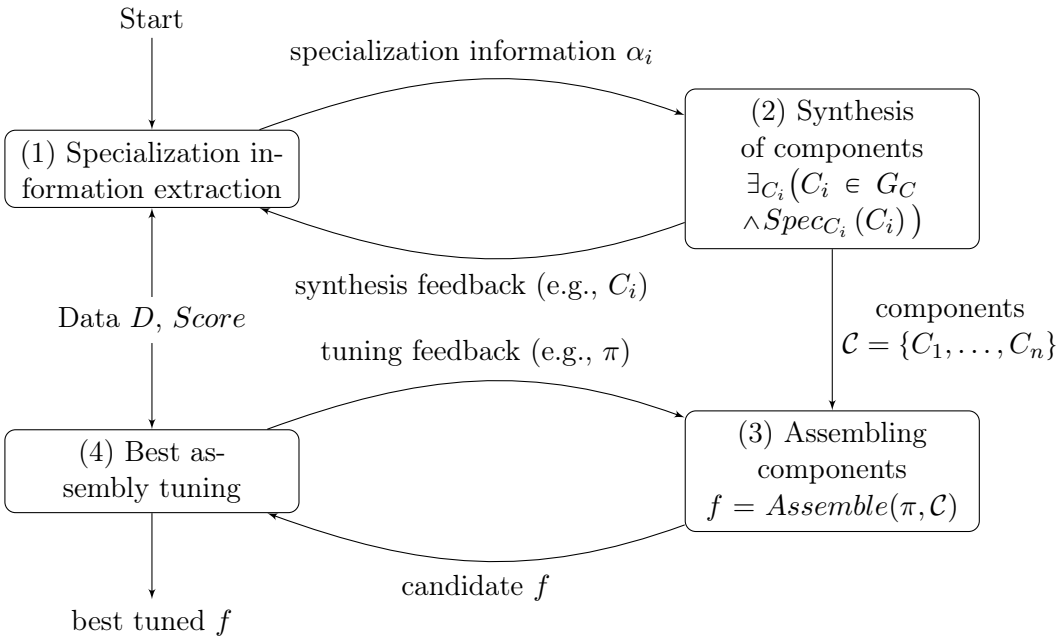

Figure 1-1: Overall synthesis/learning framework

and Sec. 1.2 (database entity matching).

1.1 Formula simplification with Swapper

Our first application domain for the overall framework is formula simplification, which 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 a problem before it is solved through some combination of abstraction refinement and theory reasoning [27, 41]. Moreover, many applications that rely on solvers often implement their own formula simplification layer to rewrite formulas before passing them to the solver [28, 31, 36].

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 domain, we address the problem of automatically learning a formula simplifier from unsupervised data (a corpus of benchmark problems or formulas) such that the performance of the solver improves by using this simplifier. We instantiate the overall framework from Fig. 1-1 as SWAPPER, a system 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.

The simplifiers 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.

To tackle these challenges, SWAPPER operates in four phases (Fig. 1-2). In the first phase (1), the system identifies the specialization information by using representative sampling. The specialization information, in this case, is a common repeating sub-term along with information about the context in which it occurs 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 (or components) which in the third phase (3) must be compiled (or assembled) 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

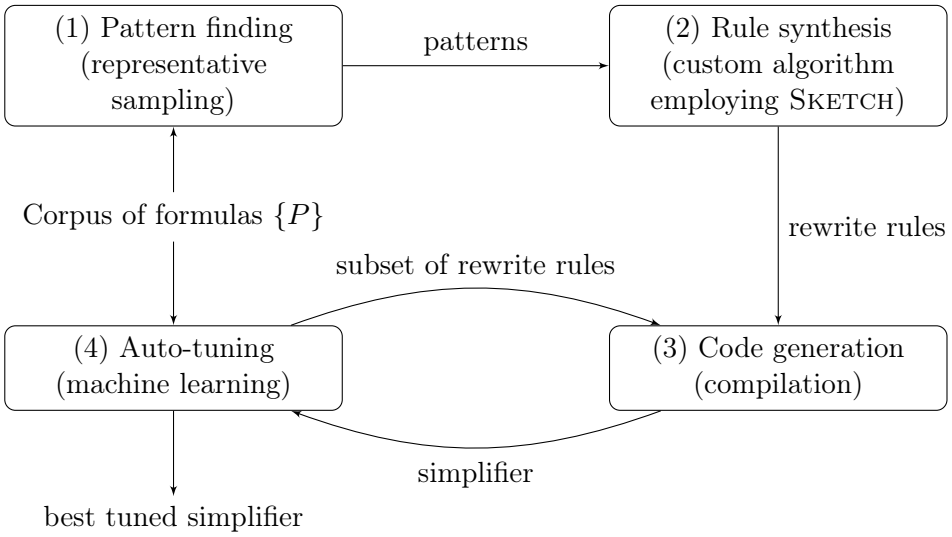

Figure 1-2: Overall phases for the SWAPPER system

on a subset of the corpus (Training set) in an effort to identify the best combination (or assembly) of the rules that maximizes solver performance. The details of the instantiation of the overall framework from Fig. 1-1 as SWAPPER are presented in Fig. 1-3.

One area where such formula simplification is particularly important is constraint-based synthesis. In particular, in this thesis, we focus SWAPPER on formulas generated by the SKETCH synthesis solver [111]. We choose the 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 [61]. 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 [102], query extraction [37], automated grading (Autograder) [100], sketching Java programs [63], SyGuS competition benchmarks [10], synthesizing optimal CNF encodings [62] and programming of line-rate switches [108]. 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 thesis, we will focus on two important domains: Autograder and SyGuS competition benchmarks. In addition, we will also present a small case study with the CNF (SAT) encodings benchmarks.

data D: an unsupervised corpus of benchmark problems or formulas

function f: a formula simplifier that simplifies a formula to another formula

$Spec(f, D)$: the function f should maintain logical equivalence of the formula that is being transformed by f for all input formulas (not necessarily only the ones in D)

grammars G, G_C: the grammar G corresponds to the grammar of C++ programs and G_C corresponds to the grammar of conditional rewrite rules

specialization information α_i: a pattern along with some static analysis information (Subsec. 2.2.1) about the context in which this pattern occurs in D

synthesis feedback: none. Pattern finding orders the patterns α_i based on their frequency of occurrence in the corpus without any synthesis feedback

component C_i: a rewrite rule $LHS(x) \xrightarrow{pred(x)} RHS(x)$

$Spec_{C_i}(C_i)$: (i) correctness of the rewrite rule C_i i.e., the evaluation of $LHS(x)$ should be the same as the evaluation of $RHS(x)$ whenever $pred(x)$ is true (ii) $LHS(x)$ should be the same as the pattern from α_i (iii) $pred(x)$ should be true in the context from α_i around $LHS(x)$

assembling components: code generation for a simplifier that applies the synthesized rewrite rules one at a time (Chapter 4)

assembly parameters π / **tuning feedback**: which subset of the synthesized rules to use and in what order to apply these rules

$Score(f, D)$: the performance of a solver using the simplifier f on problems from the corpus of benchmarks D

Figure 1-3: Instantiation of the overall framework (Fig. 1-1) as Swapper

1.2 Database entity matching with EM-Synth

Our second application domain for the overall framework is entity matching (EM), where a system or user finds records that refer to the same real-world object, is a fundamental problem of data integration and data cleaning.

There is a key tension in EM algorithms: on one hand, algorithms that properly match records are clearly preferred. On the other hand, *interpretable* EM rules – that a human can understand because they consist of a logical structure (as opposed to statistical models that are composed of weights and functional parameters) – are desirable for two reasons. First, there are a number of critical applications, such as healthcare [29] and other do-

mains [70, 71], where users need to understand *why* two entities are considered a match. Second, *interpretable* EM rules can be optimized at execution time by using blocking-based techniques [45], while such techniques are harder to apply to uninterpretable models.

Systems that use probabilistic models – such as machine learning methods based on SVMs [21], or fuzzy matching [47] – are harder to interpret than rules, and hence are often not preferred. In contrast, rule-based systems [45] offer better interpretability, particularly when the rules can be constrained to be simple with relatively few clauses. A key question is whether rules can match the effectiveness of probabilistic approaches while preserving interpretability. Intuitively, hand-writing interpretable EM rules may be practical in some limited domains. Doing so, however, is extremely time-consuming and error-prone. Hence, a promising direction is to automatically generate EM rules, by learning from positive (matching) and negative (non-matching) examples.

We present the EM-Synth system that learns EM rules (1) matching the performance of probabilistic methods and (2) producing concise rules. Our approach is to use the overall framework (Fig. 1-1) and instantiate it as the EM-Synth system (Fig. 1-6).

Example 1: Consider two tables of famous people that are shown in Figure 1-4. Dataset D_1 is an instance of schema R (name, address, email, nation, gender) and D_2 is of schema S (name, apt, email, country, sex). The EM problem is to find tuples in D_1 and D_2 that refer to the same person. Off-the-shelf schema matching tools [44, 94] may decide that name, address, email, nation, gender in table R map to name, apt, email, country, sex in table S, respectively. Given a tuple $r \in D_1$ and a tuple $s \in D_2$, a straightforward EM rule is that *the value pairs from all aligned attributes should match* such as:

$$\begin{aligned}\varphi_1\colon\ & r[\mathsf{name}] \approx_1 s[\mathsf{name}] \quad \bigwedge r[\mathsf{address}] \approx_2 s[\mathsf{apt}] \\ & \bigwedge r[\mathsf{email}] = s[\mathsf{email}] \bigwedge r[\mathsf{nation}] = s[\mathsf{country}] \\ & \bigwedge r[\mathsf{gender}] = s[\mathsf{sex}]\end{aligned}$$

Here, $\approx_1$ and $\approx_2$ are different domain-specific similarity functions. Rule φ_1 says that a tuple $r \in D_1$ and a tuple $s \in D_2$ refer to the same person (i.e., a match), if they have *similar* or *equivalent* values on all aligned attributes. □

However, in practice, the rule φ_1 above may not be able to match many similar pairs of records, since real-world data may contain multiple issues such as misspellings (e.g., s_3[name]), different formats (e.g., r_2[name] and s_1[name]), and *Null* or missing values (e.g., r_3[email] and r_4[email]). Naturally, a robust solution is to have a set of rules that

	name	address	email	nation	gender
r_1	Catherine Zeta-Jones	9601 Wilshire Blvd., Beverly Hills, CA 90210-5213	c.jones@gmail.com	Wales	F
r_2	C. Zeta-Jones	3rd Floor, Beverly Hills, CA 90210	c.jones@gmail.com	US	F
r_3	Michael Jordan	676 North Michigan Avenue, Suite 293, Chicago		US	M
r_4	Bob Dylan	1230 Avenue of the Americas, NY 10020		US	M

(a) D_1: an instance of schema R

	name	apt	email	country	sex
s_1	Catherine Zeta-Jones	9601 Wilshire, 3rd Floor, Beverly Hills, CA 90210	c.jones@gmail.com	Wales	F
s_2	B. Dylan	1230 Avenue of the Americas, NY 10020	bob.dylan@gmail.com	US	M
s_3	Micheal Jordan	427 Evans Hall #3860, Berkeley, CA 94720	jordan@cs.berkeley.edu	US	M

(b) D_2: An instance of the schema S

Figure 1-4: Sample tables for persons

collectively cover different cases.

Example 2: For example, we may have two rules as below.

φ_2: $r[\mathsf{name}] \approx_1 s[\mathsf{name}] \bigwedge r[\mathsf{address}] \approx_2 s[\mathsf{apt}]$
$\bigwedge r[\mathsf{nation}] = s[\mathsf{country}] \bigwedge r[\mathsf{gender}] = s[\mathsf{sex}]$;

φ_3: $r[\mathsf{name}] \approx_3 s[\mathsf{name}] \bigwedge r[\mathsf{email}] = s[\mathsf{email}]$

Typically, these kinds of rules are specified as disjunctions, e.g., $\varphi_2 \bigvee \varphi_3$, which indicates that a tuple $r \in D_1$ and a tuple $s \in D_2$ match, if either φ_2 or φ_3 holds. However, a more natural way, from a user perspective, is to specify the rule in a logical flow. For instance, when handling Null or missing values (e.g., $r_3[\mathsf{email}]$ and $r_4[\mathsf{email}]$), the following representation may be more user-friendly:

φ_4: **if** $(r[\mathsf{email}] \neq \text{Null} \bigwedge s[\mathsf{email}] \neq \text{Null})$

then $r[\mathsf{name}] \approx_1 s[\mathsf{name}] \bigwedge r[\mathsf{email}] = s[\mathsf{email}]$

else $r[\mathsf{name}] \approx_3 s[\mathsf{name}] \bigwedge r[\mathsf{address}] \approx_2 s[\mathsf{apt}] \bigwedge$
$r[\mathsf{nation}] = s[\mathsf{country}] \bigwedge r[\mathsf{gender}] = s[\mathsf{sex}]$

These `if-then-else` rules provide a more flexible way to model matching rules and have more expressive power than simple disjunctions. □

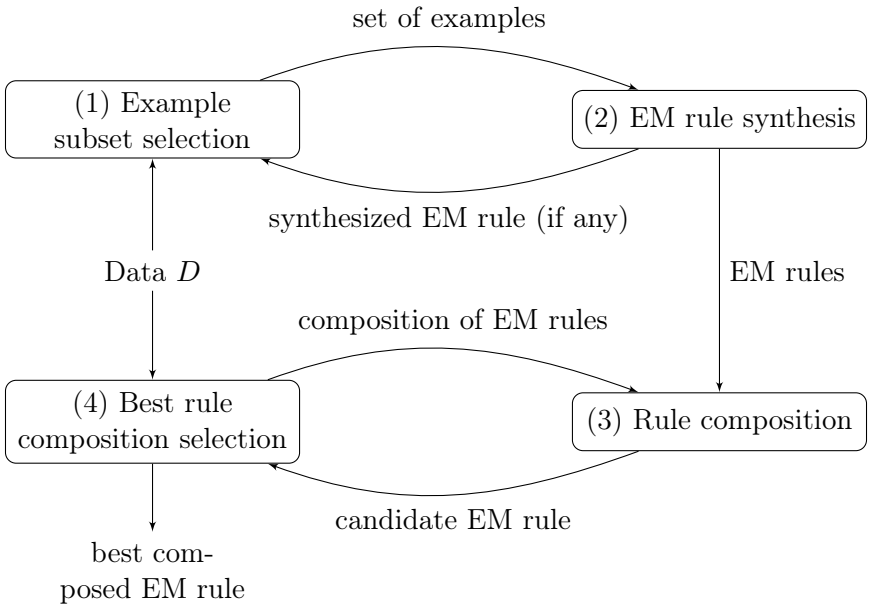

Figure 1-5: High-level description for EM-Synth framework

A high-level description of the EM-Synth system is presented in Fig. 1-5. EM-Synth synthesizes interpretable EM rules from a small set of examples and selects the next set of examples in a smart manner based on the synthesized EM rule (or the lack of one) in the first two steps (1) and (2). After generating a sizable number of EM rules, EM-Synth composes some of these rules to form a larger EM rule (step (3)). The choices of which rules to compose and how to compose them are made in step (4) in an iterative manner. The details of the instantiation of the overall framework (Fig. 1-1) as EM-Synth are presented in Fig. 1-6.

data D: a (supervised) set of positive and negative examples

function f: an EM rule that classifies an example as being positive or negative

grammars G, G_C: both of these grammars correspond to the Boolean structure of EM rules around application of similarity functions and their comparison with thresholds.

$Spec(f, D)$: true. The only logical specification for the EM rules is for them to be from the grammar G

specialization information α_i: a set of examples

synthesis feedback: the synthesized EM rule for the provided set of examples

component C_i: an EM rule

$Spec_i$: (i) correctness of the evaluation of the EM rule C_i on the provided set of examples, and (ii) the EM rule should be as small as possible with a bound on its size

assembling components: combining small EM rules to form a larger EM rule using Boolean operators from the grammar G

assembly parameters π / **tuning feedback**: which subset of the synthesized rules to use and how to combine them using operators of the grammar G

$Score(f, D)$: a numerical metric like accuracy or F-measure

Figure 1-6: Instantiation of the overall framework (Fig. 1-1) for EM-Synth

1.3 Key Contributions

In this thesis, we introduce a framework for synthesizing a function from a grammar optimizing a numerical score from large amounts of data. We summarize the high-level contributions with respect to this framework:

1. We identify a set of conditions that allow us to synthesize a complex function by solving only small local synthesis problems.

2. This framework allows optimization of a complicated numerical score over provided input data while ensuring that the logical specification of the synthesized function still holds.

3. We provide strategies and scalable infrastructure for using a general-purpose synthesizer efficiently for domain-specific problems by incorporating domain-specific knowl-

edge as a custom synthesis procedure that interacts with the general-purpose synthesizer.

This framework has been employed in three impactful application domains: (1) formula simplification in constraint solvers [106], (2) database entity matching [104], and (3) CNF encoding in SMT solvers [62]. In this thesis, we focus on the first two application domains and instantiate this framework as two systems. We summarize the key contributions for both the application domains below.

1.3.1 Automatic generation of formula simplifiers

Our major contribution for this domain is SWAPPER: a system that automatically generates efficient formula simplifiers from a corpus of benchmark problems. SWAPPER incorporates techniques from machine learning (ML), compilers and constraint-based synthesis to automate the tedious process of writing a domain-specific formula simplifier. We summarize our contributions below:

1. We automate the process of generating conditional rewrite rules specific to the common recurring patterns in formulas from a given domain by (i) using a novel algorithm for representative sampling from labeled directed acyclic graphs (DAGs) and (ii) formulating the rule generation problem as a Syntax-Guided Synthesis (SyGuS) problem.

2. We demonstrate the use of machine learning (ML) based auto-tuning to select an optimal subset and ordering of rules, and generate an efficient simplifier.

3. We extensively evaluate our approach on multiple domains from the SKETCH synthesizer and show that (i) the generated simplifiers reduce the synthesis times of SKETCH by 15%-60% relative to the existing SKETCH solver with its hand-crafted simplifier and (ii) the generated simplifiers are very domain-specific i.e., they work very well on their respective domains but perform poorly on other domains.

1.3.2 Synthesis of concise EM rules

Our major contribution in this domain is a new EM rule synthesis system called EM-Synth. We first discuss the key challenges in automatically discovering good EM rules from examples and then present our contributions targeting those challenges.

Challenges

There are two key challenges in automatically discovering good EM rules from examples.

(1) Interpretability vs. effectiveness. While interpretability is crucial in many domains, it also might sacrifice the effectiveness of the system. In real-world applications, it is often not easy, if not impossible, to find matching tuples by considering only few attribute combinations. A solution to the EM rule mining problem should keep rules simple and concise but still match the effectiveness of probabilistic approaches.

(2) Large search space. Consider two relations with n aligned attributes. There are $m = 2^n$ possible combinations of attributes. If we constrain ourselves to EM rules that consist of arbitrary selections of these attribute combinations represented in **DNF** (disjunctive normal form), this results in a search space of $\Sigma_{i=1}^{m}\binom{m}{i} = 2^m - 1 = 2^{2^n} - 1$. For instance, the search space is $2^{2^5} - 1$ (4 billion combinations) for Example 2, which contains only 5 attributes! Adding to the above complexity is that for each attribute, there is a large set of possible similarity functions, e.g., Levenshtein distance or Jaccard similarity, that could be used to determine if the attributes match. On top of that, each function also needs a similarity threshold to be compared against.

Additionally, our solution must be designed to handle the practical cases when the training data is often small or limited, and the data itself may have missing values.

EM-Synth contributions

EM-Synth generates rules that are both concise and effective. These concise rules are easy for the end user to interpret, and at the same time, they perform as well as more complicated probabilistic rules or rules written purely as DNFs (Challenge 1). This is because they are expressed in a rich grammar that includes negations and `if-then-else` clauses. Note that, similar to other database applications that rely on synthesis [97], our system uses a predefined grammar fixed across all the datasets – users do not need to know or specify the grammar.

Our approach is to instantiate the overall framework (Fig. 1-1) presented in this thesis as the EM-Synth system. EM-Synth uses a predefined grammar and a set of examples (specialization information) to synthesize EM rules (components) iteratively. EM-Synth adopts the idea of Counter-Example Guided Inductive Synthesis (CEGIS) [112] to perform synthesis

from small sets of examples, and is inspired by Random Sample Consensus (RANSAC) [49] to avoid examples that may make the algorithm under-perform. Each synthesis problem is efficiently solved with SKETCH by incorporating a special-purpose solver inside SKETCH in a novel way (Challenge 2). Moreover, EM-Synth uses some smart assembly strategies to combine the EM rules together to improve their effectiveness while maintaining their interpretability. We summarize our contributions as follows:

1. We define the problem of synthesizing EM rules from positive-negative examples. In particular, we use *General Boolean Formulas* (**GBF**) to represent EM rules. We show how to formulate this synthesis problem using the SyGuS framework [10] and use the SKETCH solver to solve these problems. Moreover, we show how to use a special-purpose solver inside SKETCH to solve these synthesis problems efficiently.

2. We describe novel additions to EM-Synth to avoid over-fitting, to eliminate biased samples, and we show how to compute the composition or assembly of multiple rules using two different techniques to improve their effectiveness.

3. We experimentally verify that our system significantly outperforms other interpretable models (i.e., decision trees with low depth, SIFI [117]) in terms of matching accuracy, even when our rules have substantially fewer clauses. It is also comparable with other uninterpretable models, e.g., decision trees with large depth, random forests, gradient tree boosting and SVM, on accuracy. We show that our approach produces very concise EM rules and uncover their superior interpretability with a user study. We show the effectiveness of our system against other interpretable models with varying amounts of (possibly small) training data. We show the large benefit in efficiency of testing some EM-Synth generated rules in practice as compared to SVM. The results also show the efficiency of training our system in producing interpretable rules.

1.4 Thesis Overview

In Chapter 2, we discuss the synthesis problems for the two application domains considered in this thesis and describe how we solve them. In Chapters 3, 4 and 5, we present the corresponding specialization information extraction, assembly of components and best assembly tuning methods used in these two application domains, respectively. In Chapter 6, we discuss

the shared infrastructure that powers efficient execution of SWAPPER and EM-Synth. In Chapters 7 and 8, we discuss the implementations of the SWAPPER and EM-Synth systems, respectively, along with their evaluations with exhaustive experiments. Chapter 9 presents relevant related work and establishes uniqueness of our systems, and Chapter 10 concludes this thesis.

Chapter 2

Synthesis of components in application domains

One of the crucial steps of the overall framework (Fig. 1-1) is the synthesis of components. The sub-problem of synthesizing components having a syntactic structure provided by a grammar and satisfying a logical specification is at the core of this step. In a lot of recent related work, researchers have used domain-specific synthesizers to efficiently solve such synthesis problems [42,52,57,88,114,119], but they have had to build synthesis infrastructure from scratch for each application. In this thesis, we perform synthesis with a general-purpose synthesizer (Sketch [111]) and incorporate domain-specific insights from our application domains to make the synthesis procedure more efficient. This customization with a general-purpose solver (Sketch) enables us to reuse a lot of infrastructure already built in Sketch.

In both instantiations of the overall framework (Swapper and EM-Synth), the synthesis-of-components step corresponds to synthesizing a rule (a conditional rewrite rule in Swapper and an EM rule in EM-Synth) having a logical structure provided by a grammar and constraints that arise from the application domain and the input data. In this chapter, we discuss the synthesis problems at the core of these systems (Swapper in Sec. 2.2 and EM-Synth in Sec. 2.3) and describe how to formulate these in the well-studied syntax-guided synthesis (SyGuS) framework [10] (Sec. 2.1). In Appendix A, we also show how to solve these SyGuS problems using Sketch, an open source SyGuS solver.

2.1 Syntax-guided synthesis (SyGuS) framework

The SyGuS framework is employed to specify certain program-synthesis problems with syntactic constraints in a standardized format [10]. Such a synthesis problem consists of finding a program p such that a constraint C capturing the correctness of p is satisfied. This synthesis problem is further constrained in three ways: (1) the universe of possible programs p is restricted to syntactic expressions described by a grammar G, (2) the logical symbols and their interpretation are restricted to a background theory $\mathcal{T}$ for which well-understood decision procedures are available for determining satisfaction modulo $\mathcal{T}$, and (3) the constraint C is limited to a quantifier-free first-order formula in the background theory $\mathcal{T}$.

SyGuS problems can be represented abstractly as a *grammar* G representing a set of expressions or programs, together with a constraint C on the behavior of the desired expression. A *grammar* G is a set of recursive rewriting rules (or *productions*) used to generate expressions over a set of *terminal symbols* and *non-terminal (recursive) symbols*. The productions provide a way to *derive* expressions from a designated *initial symbol* by applying the productions one after another. An example SyGuS problem with a grammar and an associated constraint is given below:

$$\begin{array}{rl} \mathsf{grammar} & expr \rightarrow expr \vee expr \ (\mathbf{bound} : B) \\ & expr \rightarrow x \mid y \mid \neg x \mid \neg y \\ \mathsf{constraint} & \big(x \implies (\neg y = expr)\big) \wedge \big(y \implies (\neg x = expr)\big) \end{array}$$

Figure 2-1: An example grammar of a simple Boolean expression

The above grammar has a non-terminal symbol (also the initial symbol) *expr* that represents a disjunction of variables x, y or their negations. The underlying theory corresponds to the theory of a Boolean algebra with two atomic variables x and y i.e., x and y are Boolean variables that can take the constant values 1 ($true$) and 0 ($false$). The above representation of the constraint assumes that all terminal symbols and variables are universally quantified i.e., the constraint should be satisfied for all values taken by the terminal symbols/variables (x and y in this example). The space of expressions is bounded because of a parameter B that bounds the number of times a production can be used.

A SyGuS solver would take the above grammar and constraint as an input and output a candidate expression *expr* from the grammar that satisfies the constraint (if there exists one) e.g., if the bound $B = 1$ then the solver may output the expression $\neg x \vee \neg y$.

Note that in the example SyGuS problem above, the constraints allow us to easily evaluate the required output function on all possible values of x and y. The purpose of solving such a SyGuS problem would be to find a concise representations of this function from a restricted grammar. In general, the constraints and the grammar can be more complicated and hard to reason about manually. Moreover, the problem may also be underspecified i.e., there may be multiple possible functions in the provided grammar that satisfy the given constraints. An example underspecified SyGuS problem is given below:

$$
\begin{aligned}
\textsf{grammar} \quad & expr \rightarrow \textbf{ if } boolExpr \textbf{ then } expr \textbf{ else } expr \ (\textbf{bound} : B_1) \\
& expr \rightarrow expr + expr \mid expr \times expr \ (\textbf{bound} : B_2) \\
& expr \rightarrow a \mid b \\
& boolExpr \rightarrow expr \ \ compOp \ \ expr \\
& compOp \rightarrow \ > \ \mid \ \geqslant \ \mid \ = \ \mid \ \neq \\
\textsf{constraint} \quad & (expr \geqslant a) \wedge (expr \geqslant b)
\end{aligned}
$$

A solution of this SyGuS problem is a function from the provided grammar that can be used as an upper bound on two integer inputs a and b. Some possible solutions are the expressions: $a + b$, $a \times a + b \times b$, and **if** $(a > b)$ **then** a **else** b. For this problem, a SyGuS solver will output one of these solutions given appropriate bounds B_1 and B_2.

2.2 Synthesis of conditional rewrite rules in Swapper

Following the overall framework, in Swapper (Fig. 1-2), the problem of automatically generating the formula simplifier is broken down into multiple synthesis problems. Each of these problems correspond to synthesis of conditional 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: $\forall x \ \big(pred(x) \implies (LHS(x) = RHS(x))\big)$.

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) i.e.,

the pattern $or(lt(a,b), lt(a,d))$ with the smaller pattern on the right hand side (RHS).

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

Multiple conditional rewrite rules can be used to simplify large logical formulas by rewriting various parts of the formula locally as described in Subsec. 2.2.1. We describe the core rule-synthesis problem in Subsec. 2.2.2. Since we focus on the SKETCH solver as the target of formula simplification in this thesis, we constrain the structure of LHS, RHS and *pred* based on the internals of SKETCH (Subsec. 2.2.3).

2.2.1 Formula simplification in SKETCH

We assume that the rewriter (while applying rewrite rules) 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.

SWAPPER identifies promising LHS patterns for rules in the pattern-finding phase (more details in Sec. 3.2), together with properties of their free variables that can be assumed to hold in the contexts where these LHS patterns appear. For example, SWAPPER may discover that the pattern $or(lt(a,b), lt(a,d))$ is very common, so finding a rewrite 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 $b \leqslant 0$ and $d > 0$. The next challenge is to synthesize the rewrite rules for this pattern that will be applicable in such contexts.

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 SyGuS problem [10] (Subsec. 2.2.2) and solve it using the SKETCH synthesis tool [111] (Appendix A).

2.2.2 Core rule-synthesis problem

As mentioned above, SWAPPER finds rules from patterns and the context information collected during the pattern-finding phase (we will explain more details of the pattern-finding phase in Sec. 3.2). The motivation behind using the context information is (1) to ensure that all generated rules are applicable in the corpus and (2) to constrain the synthesis problem to reduce the size of the search space and efficiently solve the synthesis problem.

Now, we describe how this context information is used for rule synthesis. SWAPPER finds corresponding rewrite rules for the set of patterns $\{Q\}$ obtained from the pattern-finding phase. Each of these patterns Q is a sub-term of some formula P from the corpus of formulas $\{P\}$ provided as an input to SWAPPER i.e., the pattern Q occurs as a rooted subgraph in the formula P (represented as a directed acyclic graph or DAG). The inputs or free variables $x = (x_1, x_2, \ldots, x_n)$ of Q correspond to sub-terms $(a_1, a_2, \ldots, a_n)$ of P such that the sub-term a_i is rooted at the input x_i of Q in P. Fig. 2-2 shows an example DAG P with a pattern Q (marked by the dotted ellipse), inputs $(x_1, x_2, \ldots, x_6)$ (arrows marked in blue) of Q and the corresponding sub-terms $(a_1, a_2, \ldots, a_n)$ (marked in orange).

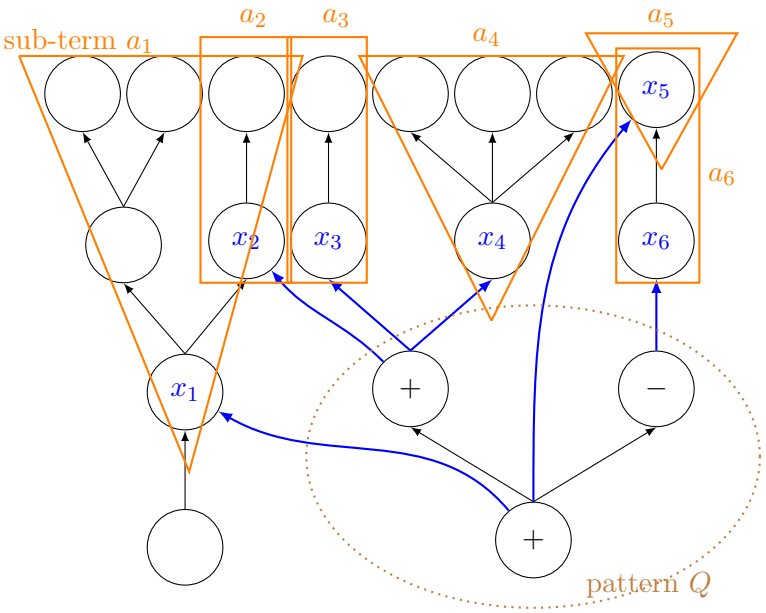

Figure 2-2: Relationship between inputs $(x_1, x_2, \ldots, x_n)$ of a pattern Q and the corresponding sub-terms $(a_1, a_2, \ldots, a_n)$ of the formula P

Using the *static* function discussed in Subsec. 2.2.1, for each pattern $Q(x)$, the pattern-finding phase also collects a list of *static* range values of the sub-terms $(a_1, a_2, \ldots, a_n)$

discussed above. These values are collected for each context in which the pattern Q occurs in the corpus. More formally, the pattern-finding phase will collect a pattern $Q(x)$ with inputs $x = (x_1, x_2, \ldots, x_n)$ along with the lists

$$\Big(static(a_1^j), static(a_2^j), \ldots, static(a_n^j)\Big) \ \forall \ 1 \leqslant j \leqslant m$$

where m is the number of occurrences of Q in the corpus and $\Big(a_1^j, a_2^j, \ldots, a_n^j\Big)$ are the subterms corresponding to the inputs x of $Q(x)$ for its j^{th} occurrence in the corpus. In the rule-synthesis phase, we construct the predicates $assume_j(x)$ over the free variables x that evaluate to true when each free variable x_i is in the range of values given by $static(a_i^j)$ i.e.,

$$assume_j(x) = \bigwedge_{i=1}^{n} \Big(x_i \in static(a_i^j)\Big)$$

We use the notation $assume(x)$ to denote the collection of all $assume_j(x)$'s i.e.,

$$assume(x) = \Big(assume_1(x), assume_2(x), \ldots, assume_m(x)\Big)$$

$assume(x)$ is used to constrain the predicate $pred(x)$ for the conditional rewrite rule to be synthesized. Intuitively, if the predicate $pred(x)$ satisfies the property that for some j, $\forall x \ \big(assume_j(x) \implies pred(x)\big)$ then this guarantees that the rewrite rule

$$Q(x) \xrightarrow{pred(x)} RHS(x)$$

will be applicable at least at the j^{th} occurrence of the pattern Q. This way we can avoid rules with predicates that will never hold in practice and focus on the rules with predicates that are implied by some assumptions $assume_j(x)$ obtained from the pattern-finding phase.

Apart from finding applicable predicates, Swapper also needs to find correct and effective rewrite rules for a given LHS pattern. We incorporate these requirements as constraints and present the formal description of the core rule-synthesis problem for Swapper in Problem 3.

Problem 3 *Given a pattern $LHS(x)$, collection of predicates $assume(x)$ discovered by* Swapper *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 con-*

straints:

1. $\bigvee_{j=1}^{m} \Big(\forall x : assume_j(x) \implies pred(x)\Big)$
2. $\forall x : pred(x) \implies (LHS(x) = RHS(x))$
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 the previous constraints*
5. *for a pred*(x) *and* $RHS(x)$ *that satisfy the above constraints, size*(RHS) *should be as small as possible.*

Note that there can be multiple instances of *pred*(x) that are the weakest possible (item 4. from Problem 3) in terms of the permissiveness. For example, the following two rules below (Rules 1 and 2) have the weakest predicates when the predicate space is given by the following grammar:

$$\begin{aligned} \textsf{grammar} \quad & pred \rightarrow var < var \\ & pred \rightarrow true \\ & var \rightarrow a \mid b \mid d \end{aligned}$$

Rule 1:

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

Rule 2:

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

i.e., there is no other predicate *pred*(a, b, d) in the above predicate grammar for which we can find a rewrite rule satisfying the constraints $1 - 3, 5$ from Problem 3 along with constraint 4 that translates to:

$$\forall a, b, d \ \ ((b < d \implies pred(a,b,d)) \wedge (a < b \implies pred(a,b,d)))$$

Note that the predicate *true* is implied by both of the predicates $b < d$ and $a < b$ but, as we will see below in Subsec. 2.2.3, there is no $RHS(a, b, d)$ in the RHS grammar with a

size smaller than $size(LHS)$ that will make the following rewrite rule valid:

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

2.2.3 Space of expressions and predicates

SKETCH, like most solvers, represents constraints as directed acyclic graphs (DAGs) in order to exploit sharing of sub-terms within a formula. The formulas in SKETCH consist of 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 universally and existentially quantified variables: *inputs* and *controls* respectively. The formula-simplification pass that is the subject of this work 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) [111]. The complete list of terms supported by SWAPPER is shown in Fig. 2-3.

$e =$	$boolop(e_1, e_2)$	apply a Boolean binary operator on e_1, e_2
		$e.g.$, and, or, xor
	$neg(e_1)$	Boolean unary negation operator representing $\neg e_1$
	$arithop(e_1, e_2)$	apply an arithmetic operator on e_1, e_2
		$e.g.$, $plus, times, mod, lt, gt$
	$minus(e_1)$	arithmetic unary minus operator representing $-e_1$
	$inp(id)$	universally quantified variable (input)
	$ctrl(id)$	existentially quantified variable (control)
	$arr_r(e_i, e_a)$	read at index e_i in array e_a
	$arr_w(e_i, e_a, e_v)$	write at index e_i value e_v in array e_a
	$arr_{create}(c)$	new array with a default value c: $\{\_ \mapsto c\}$
	$n(c)$	integer/Boolean constant with value c
	$mux(e_c, e_0, \ldots e_i)$	multiplexer chooses based on value e_c
	$assert(e)$	assertion of a Boolean expression e

Figure 2-3: The language of formula expressions in SKETCH.

The space for RHS is specified by a template that simulates the computation of a function using temporary variables. This computation can be naturally interpreted as a

pattern. The essential template for the generator for RHS is shown here:

$$\begin{aligned} RHS(x) \equiv \quad & let \quad && t_1 = \textbf{simpleOp}(x); \\ & && t_2 = \textbf{simpleOp}(x, t_1); \\ & && \ldots \\ & && t_k = \textbf{simpleOp}(x, t_1, ..., t_{k-1}); \\ & in\ t_k \end{aligned}$$

where **simpleOp** represents a single operation node (e.g. *and*, *plus* etc. from Fig. 2-3) with its operands being selected from the arguments. For example, the expression $(a+b) \times c$ can be represented as:

$$\begin{aligned} \text{let} \quad & t_1 = plus(a, b); \\ & t_2 = times(t_1, c); \\ \text{in} \quad & t_2 \end{aligned}$$

We put a strict upper bound on k as one less than the number of nodes in the fixed LHS.

For *pred*, we employ a simple Boolean expression grammar that considers conjunctions of equalities and inequalities among variables:

$$\begin{aligned} \textsf{grammar} \quad & pred(x) \rightarrow boolExpr(x) \\ & pred(x) \rightarrow boolExpr(x) \wedge boolExpr(x) \\ & boolExpr(x) \rightarrow x_i \text{ binop } x_j \\ & boolExpr(x) \rightarrow \neg x_i \mid x_i \\ & where \text{ binop } \in \{<, >, =, \neq, \leqslant, \geqslant\} \\ & x = (x_1, x_2, \ldots, x_n), 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant n \end{aligned}$$

These predicates are inspired by existing predicates present in the rules in SKETCH's hand-crafted formula simplifier.

Now, we are ready to discuss the details of a new hybrid approach to synthesis of rewrite rules in SWAPPER.

2.2.4 Hybrid enumerative/symbolic synthesis in SWAPPER

To solve Problem 3, SWAPPER can use SKETCH to find rewrite rules that satisfy constraints $1-3,5$ (discussed in detail later in appendix A.2) for a given $LHS(x)$. In practical terms, with many assumptions $assume(x)$ on a given $LHS(x)$, pushing these constraints to SKETCH makes it inefficient to run SKETCH instances with large $assume(x)$ constraints. We describe below a way to separately satisfy the constraints $1,4$ (from Problem 3) and use the SKETCH solver to satisfy only constraints $2,3,5$ (from Problem 3). SWAPPER breaks the full synthesis problem (all constraints from Problem 3) into two parts:

(1) Constraints and optimizations on predicates

SWAPPER uses the enumerative approach. It generates all possible candidate predicates from the specification grammar and checks for their validity based on collected assumptions $assume(x)$. It also 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 $\{a > b, \neg b, a \geqslant b, a \neq b\}$.

SWAPPER enumerates all predicates that satisfy the collected assumptions $assume(x)$ from the provided grammar as a list (allPreds in Algorithm 1) and creates a directed graph $G_{\Rightarrow}$ (ImplicationGraph in Algorithm 1) such that: (1) each predicate is a node of the graph (2) for two predicates $p_1(x), p_2(x)$ there is a directed edge from the node corresponding to $p_1(x)$ to the node corresponding to $p_2(x)$ iff $\forall x, p_1(x) \implies p_2(x)$. SWAPPER computes this relationship between predicates based on whether one predicate implies the other or not for all values of the free variables e.g. $(a < b)$ implies $a \neq b$ but doesn't imply $a > b$.

The full enumerative algorithm is shown in Algorithm 1. SWAPPER evaluates three functions on this graph $G_{\Rightarrow}$:

1. isEmpty($G_{\Rightarrow}$): this function returns *true* if there is at least one node (predicate) left in the graph $G_{\Rightarrow}$.

2. sampleLeaf($G_{\Rightarrow}$): each leaf in the graph (a node with no incoming edges) corresponds to one of the least applicable predicates. This function randomly chooses a leaf node and returns it.

Algorithm 1: Hybrid enumerative/symbolic SKETCH based approach

input : LHS(x), allPreds
output: valid rewrite rules of the form $(LHS, pred_i, RHS_i)$

```
1  G_⇒ ← ImplicationGraph(allPreds)
2  while ¬isEmpty(G_⇒) do
3      pred ← sampleLeaf(G_⇒)
4      RHS ← SKETCH(LHS, pred)
5      if RHS = NULL then
6          foreach pred' ∈ G_⇒ ∧ pred ⇒ pred' do
7              removePred(pred', G_⇒)
8      else
9          removePred(pred, G_⇒)
10         Output rule (LHS, pred, RHS)
```

3. removePred($pred, G_{\Rightarrow}$): this function removes the node corresponding to a predicate *pred* in the graph $G_{\Rightarrow}$ including all edges leading to it or originating from it.

SWAPPER iteratively finds RHS (using SKETCH as discussed below) for the least applicable predicates at any given stage. When there is no possible rewrite rule for one of the least applicable predicates *pred* then SWAPPER prunes out all predicates *pred'* implied by *pred* because there cannot exist a rule with a more applicable predicate *pred'* when there is no rule with predicate *pred*. This helps SWAPPER reduce overall time for the rule-synthesis step by selectively running a few SKETCH problem instances.

(2) Synthesis of RHS given LHS and *pred*

SWAPPER hard-codes the predicate *pred* and realizes the RHS synthesis problem in SKETCH using the **generator** and **minimize** features [105, 110] of the SKETCH language (also described later in appendix A.2). The result of solving this SKETCH instance provides the smallest RHS such that the rule $(LHS, pred, RHS)$ is valid for a fixed LHS and *pred* as required by Algorithm 1.

This concludes our discussion of synthesis of conditional rewrite rules in SWAPPER. Now, we discuss the synthesis of Entity Matching (EM) rules that corresponds to the synthesis of components in the EM-Synth system (Fig. 1-5).

2.3 Synthesis of EM rules in EM-Synth

Entity Matching (EM) rules, similar to conditional rewrite rules, have a syntactic structure given by a grammar. And, similar to SWAPPER, we use SKETCH to synthesize EM rules in EM-Synth.

As shown in Examples 1 and 2 (Sec. 1.2), entity matching rules considered in this thesis have a Boolean structure around some *atomic* predicates. Each atomic predicate is evaluated based on similarity of values from two schema-matched columns (or attributes). We formalize the notations required to describe the structure of these EM rules (Subsec. 2.3.1), discuss the core syntax-guided synthesis problem (Subsec. 2.3.2) at the heart of EM-Synth system in this section and how to solve it using a custom synthesizer inside SKETCH (Subsec. 2.3.3).

2.3.1 Notation and EM-GBF rule-synthesis problem

Let $R[\mathsf{A}_1, \mathsf{A}_2, \ldots, \mathsf{A}_n]$ and $S[\mathsf{A}'_1, \mathsf{A}'_2, \ldots, \mathsf{A}'_n]$ be two relations with corresponding sets of n aligned attributes A_i and A'_i ($i \in [1, n]$). We assume that the attributes between two relations have been aligned and provided as an input; this can be either manually specified or done automatically using off-the-shelf schema matching tools [18]. Note that our approach naturally applies to the deduplication scenario with only one relation, i.e., $R = S$ where no schema alignment is needed ($\mathsf{A}_i = \mathsf{A}'_i, \forall i \in [1, n]$). In the following, we will discuss the notation with two relations. We use datasets both with two relations and a single relation for experiments.

Let r, s be records in R, S and $r[\mathsf{A}_i], s[\mathsf{A}'_i]$ be the values of the attribute $\mathsf{A}_i, \mathsf{A}'_i$ in records r, s, respectively.

Similarity functions

A *similarity function* $f(r[\mathsf{A}_i], s[\mathsf{A}'_i])$ computes a similarity score in the real interval $[0, 1]$. A bigger score means that $r[\mathsf{A}_i]$ and $s[\mathsf{A}'_i]$ have a higher similarity. Examples of similarity functions are cosine similarity, edit distance, and Jaccard similarity. A library of similarity functions $\mathcal{F}$ is a set of such general-purpose similarity functions, for example the Simmetrics (`https://github.com/Simmetrics/simmetrics`) Java package.

Now we define rules for *matching* records $r \in R$ and $s \in S$.

Attribute-matching rules

An *attribute-matching rule* is a triple $\approx(i, f, \theta)$ representing a Boolean function with value $f(r[\mathsf{A_i}], s[\mathsf{A'_i}]) \geqslant \theta$, where $i \in [1, n]$ is an index, f is a similarity function and $\theta \in [0, 1]$ is a threshold value. Attribute-matching rule $\approx(i, f, \theta)$ evaluating to *true* means that $r[\mathsf{A_i}]$ *matches* $s[\mathsf{A'_i}]$ relative to the specific similarity function f and threshold θ.

We use the notation $r[\mathsf{A_i}] \approx_{(f,\theta)} s[\mathsf{A'_i}]$ to denote an attribute-matching rule for some underlying similarity function f and threshold θ. We will simply write $r[\mathsf{A_i}] \approx s[\mathsf{A'_i}]$ when it is clear from the context.

Record-matching rules

A *record-matching rule* is a conjunction of a set of attribute-matching rules on different attributes. Intuitively, two records r and s *match* iff all attribute-matching rules in the set evaluate to *true*.

Disjunctive matching rule

A *disjunctive matching rule* is a disjunction of a set of record-matching rules. Records r and s are matched by this rule *iff* they are matched by at least one of this rule's record-matching rules.

Indeed, a *disjunctive matching rule* can be seen as a formula in Disjunctive Normal Form (**DNF**) over *attribute-matching rules* as:

$$\bigvee_{p=1}^{P} \left(\bigwedge_{q=1}^{Q_p} \approx(i_{(p,q)}, f_{(p,q)}, \theta_{(p,q)}) \right)$$

where P is the number of record-matching rules and $(Q_1, \ldots, Q_P)$ are the number of attribute-matching rules in each of the P record-matching rules, respectively.

There are two main shortcomings of using **DNF** rules:

(1) [Not Concise.] A **DNF** $(u_1 \bigwedge v_1) \bigvee (u_1 \bigwedge v_2) \bigvee (u_2 \bigwedge v_1) \bigvee (u_2 \bigwedge v_2)$ is equivalent to a much more concise formula $(u_1 \bigvee u_2) \bigwedge (v_1 \bigvee v_2)$.

(2) [Expressive Power.] A **DNF** rule without negations cannot express the logic "if (u) then (v) else (w)", which can be modeled using a formula such as $(u \bigwedge v) \bigvee (\neg u \bigwedge w)$. Traditionally, negations are not used in tools that generate EM rules with a Boolean structure.

Hence, a more natural way than **DNF** to define ER rules is to use general Boolean formulas, as defined below.

Boolean formula matching rule

A *Boolean formula matching rule* is an arbitrary *Boolean formula* over attribute-matching rules as its variables and conjunction ($\bigwedge$), disjunction ($\bigvee$) and negation ($\neg$) as allowed operations.

We formulate a *Boolean formula matching rule* as a *general Boolean formula* (**GBF**).

Example 3: Consider Example 2. Let the similarity function for matching attributes name in R and name in S (resp. address in R and apt in S) be Levenshtein (resp. Jaccard), with threshold 0.8 (resp. 0.7).

[Attribute-matching rule.] $r[\mathsf{name}] \approx s[\mathsf{name}]$ can be formally represented as $\approx(1, \texttt{Levenshtein}, 0.8)$, where the number 1 is the positional index for the 1^{st} pair of aligned attributes, i.e., attributes (name, name) for relations (R, S).

[Record-matching rule.] φ_2 can be formalized as:

$$\varphi_2 : \approx(1, \texttt{Levenshtein}, 0.8) \bigwedge \approx(2, \texttt{Jaccard}, 0.7)$$
$$\bigwedge =(4, \texttt{Equal}, 1.0) \qquad \bigwedge =(5, \texttt{Equal}, 1.0)$$

Similarly, φ_3 can be formalized as:

$$\varphi_3 : \approx(1, \texttt{Levenshtein}, 0.8) \bigwedge =(3, \texttt{Equal}, 1.0)$$

[Disjunctive matching rule.] A disjunctive matching rule for φ_2 and φ_3 is the disjunction of the above two record-matching rules, $\varphi_2 \bigvee \varphi_3$.

[Boolean formula matching rule.] Consider a custom *similarity* function `noNulls` that returns 1.0 when the values of the corresponding attributes are both not null and 0.0 otherwise. Using this function, we can formalize φ_4 as:

φ_4: **if** $(\approx(1, \texttt{noNulls}, 1.0))$ **then** φ_2 **else** φ_3

□

There are two reasons why we propose to synthesize **GBF** rules instead of **DNF** rules.

1. **GBF** can concisely represent a **DNF** and increase its expressibility thereby enhancing the readability.

2. Traditionally used EM rules in the **DNF** [117] form require each attribute to show

up with the same similarity function and threshold everywhere in the **DNF**, with the main purpose of reducing the search space of their solution.

In our **GBF** rules, we allow one attribute to have different similarity functions in the *Boolean formula*, since values in the same column are not always homogeneous, and we need different similarity functions to capture different matching cases. Consider for instance the attribute name. In rule φ_2, the similarity function used is Levenshtein with threshold 0.8. A variant φ_3' of φ_3 could use Jaccard similarity with threshold 0.6 for name.

Now, we are ready to discuss the rule-synthesis problem in EM-Synth.

EM-GBF rule-synthesis problem

In the EM-Synth system (Fig. 1-5), the synthesis-of-components step corresponds to synthesizing EM rules from a grammar using a set of examples (the specialization information extracted from the data) i.e., this synthesis step constrains the grammar to produce rules that are satisfied on all examples in the set. This set of examples, denoted by $\mathbf{E}_{\text{SYN}} = \mathbf{M}_{\text{SYN}} \cup \mathbf{D}_{\text{SYN}}$, consists of potentially some positive examples $\mathbf{M}_{\text{SYN}}$ (*matching*), i.e., pairs of records that represent the same entity, and some negative examples $\mathbf{D}_{\text{SYN}}$ (*different*), i.e., pairs of records that represent different entities.

Problem 4 *Given the sets $\mathbf{M}_{SYN}$ and $\mathbf{D}_{SYN}$ of positive and negative examples and a grammar $G_{\mathbf{GBF}}$, the* **EM-GBF** *rule-synthesis problem is to discover a* **GBF** φ *that satisfies all examples in $\mathbf{E}_{SYN} = \mathbf{M}_{SYN} \cup \mathbf{D}_{SYN}$.*

In the rest of this section, we will discuss how to solve Problem 4. We start with a description of the SyGuS formulation for this problem below.

2.3.2 Core SyGus Formulation

In this subsection, we describe how to formulate the **EM-GBF** rule-synthesis problem as a SyGuS problem [10] with a grammar and corresponding constraints (Sec. 2.1). In the context of EM, the grammar represents the space of **GBF**s, and the constraints correspond to satisfaction of the set of user-supplied examples provided as an input. We provide the precise grammar and constraints to formulate the **EM-GBF** rule-synthesis problem below.

Grammar for EM-GBF rule-synthesis problem

In order to formulate the **EM-GBF** rule-synthesis problem in the SyGuS framework, we use a generic Boolean formula grammar ($G_{\mathbf{GBF}}$) defined below:

$$
\begin{aligned}
&\textsf{grammar} \quad G_{\textsf{attribute}} \rightarrow r[\mathsf{A_i}] \approx_{(f,\theta)} s[\mathsf{A'_i}] \\
&\qquad\qquad i \in [1,n]; f \in \mathcal{F}; \theta \in [0,1] \\
&\textsf{grammar} \quad G_{\mathbf{GBF}} \rightarrow G_{\textsf{attribute}}\ (\mathbf{bound} : N_a) \\
&\qquad\qquad \left.\begin{aligned} G_{\mathbf{GBF}} &\rightarrow \neg G_{\mathbf{GBF}} \\ G_{\mathbf{GBF}} &\rightarrow G_{\mathbf{GBF}} \wedge G_{\mathbf{GBF}} \\ G_{\mathbf{GBF}} &\rightarrow G_{\mathbf{GBF}} \vee G_{\mathbf{GBF}} \end{aligned}\right\} (\mathbf{depth} : N_d)
\end{aligned}
$$

The grammars $G_{\texttt{attribute}}$ and $G_{\mathbf{GBF}}$ represent an attribute-matching rule and a Boolean formula matching rule (**GBF**), respectively. Note that the search space represented by the above grammars is infinite because there are infinitely many real values for $\theta \in [0,1]$. We tackle this by using a special-purpose synthesizer inside SKETCH (described later in Subsec. 2.3.3). The bounds N_a and N_d make the search space for the Boolean formula finite by bounding the number of attribute-matching rules ($G_{attribute}$) in $G_{\mathbf{GBF}}$ and the depth of the expansion of the grammar, respectively.

Handling Null values in $G_{\mathbf{GBF}}$: Null (missing) values are problematic because we cannot know whether two records match on some attribute A if one record has a Null value for A. Rather than assuming that such records do not match (as was done in previous work [116]), we learn different rules for the Null and noNull case. We specify a new grammar production in $G_{\mathbf{GBF}}$ for deriving **GBF**s that capture this intuition:

$$
\begin{aligned}
&\textsf{grammar} \quad G_{\mathbf{GBF}} \rightarrow \ \mathbf{if}\ (\approx(i, \texttt{noNulls}, 1.0)) \\
&\qquad\qquad\qquad\qquad\ \ \mathbf{then}\ (G_{\mathbf{GBF}})\ \mathbf{else}\ (G_{\mathbf{GBF}}) \\
&\qquad\qquad\qquad\qquad\ \ i \in [1,n]
\end{aligned}
$$

It says that if there are no nulls in the matching attributes in a pair of records, then we should use one **GBF**; otherwise we should use a different **GBF**. This makes it possible for the synthesizer to quickly find rules similar to example φ_4 (Sec. 1.2). Note that this addition does not affect the expressibility of the grammar and is purely for making the grammar $G_{\mathbf{GBF}}$ and the synthesis process more targeted towards databases with large numbers of nulls.

Constraints for EM-GBF rule-synthesis problem

A candidate φ selected from the grammar $G_{\mathbf{GBF}}$ can be interpreted as a Boolean formula. Given both positive ($\mathbf{M}_{\text{SYN}}$) and negative ($\mathbf{D}_{\text{SYN}}$) examples, the SyGuS constraints are specified as the evaluation of this **GBF** on the provided examples being consistent:

$$\begin{aligned} &\mathsf{constraint} \quad \varphi(r_m, s_m) = \mathsf{true},\ \forall\ (r_m, s_m) \in \mathbf{M}_{\text{SYN}} \\ &\mathsf{constraint} \quad \varphi(r_d, s_d) = \mathsf{false},\ \forall\ (r_d, s_d) \in \mathbf{D}_{\text{SYN}} \end{aligned}$$

Next, we present the details of the special-purpose synthesizer built to interact with SKETCH to efficiently synthesize EM rules.

2.3.3 Numerical search for EM thresholds in SKETCH

We focus on the **EM-GBF** rule-synthesis problem of searching for a candidate **GBF** from the bounded grammar $G_{\mathbf{GBF}}(N_a, N_d)$ that satisfies all the constraints arising from examples in $\mathbf{E}_{\text{SYN}}$. The SKETCH solver is not suitable to solve these problems directly. In general, SKETCH works by analyzing every part of the grammar and constraints symbolically, reducing the search problem to a Boolean satisfiability (SAT) problem. Using SKETCH directly for this problem is impractical because it involves reasoning about complicated numerical functions. For solving this problem with SKETCH, we use a new technique that allows SKETCH to collaborate with a custom solver that handles analysis of similarity functions and synthesizes thresholds while SKETCH makes discrete decisions for the **GBF**. Specifically, SKETCH makes the decisions for (1) expanding the $G_{\mathbf{GBF}}$ grammar with multiple *atoms* or attribute-matching rules, (2) choosing examples in $\mathbf{E}_{\text{SYN}}$ to be positive ($\mathbf{E}_+$) or negative ($\mathbf{E}_-$) for each atom of the expanded **GBF**, (3) choosing the attributes $i \in [1, n]$ and similarity functions $f \in \mathcal{F}$ to be used in these atoms. The custom solver finds a numerical threshold that separates the positive ($\mathbf{E}_+$) and negative examples ($\mathbf{E}_-$) chosen by SKETCH for an atom, if one exists. Otherwise, it will ask the SKETCH solver to backtrack and make alternative discrete decisions. This solver will be called multiple times inside SKETCH. Its pseudocode is given in Algorithm 2.

As an optimization, to avoid recomputing numerical functions in the special-purpose solver, we enumerate and memoize the function evaluations on all possible values that can be obtained from aligned attributes in the examples. For example, if we have just one

example $e_1 = (r, s)$ with

$$r \equiv \{\mathsf{name} = \text{'C. Zeta-Jones'}, \mathsf{gender} = \text{'F'}\}$$
$$s \equiv \{\mathsf{name} = \text{'Catherine Zeta-Jones'}, \mathsf{sex} = \text{'F'}\}$$

then we evaluate the `Jaccard` similarity function on aligned attributes and provide the following table `evalSimFn` to the custom solver (Algorithm 2):

example id	matched attribute	function	evaluation
e_1	name|name	`Jaccard`	0.5
e_1	gender|sex	`Jaccard`	1.0

Algorithm 2: Custom synthesizer for EM-Synth inside SKETCH

input : f : chosen similarity function
a : matched attribute Id
$\mathbf{E}_+$: examples chosen to be positive
$\mathbf{E}_-$: examples chosen to be negative
`evalSimFn` : similarity function evaluation table

output: *exists* : does a valid threshold exist
θ : a valid threshold separating $\mathbf{E}_+$ & $\mathbf{E}_-$

$\theta_{atmost} \leftarrow 1.0$
for $e \in \mathbf{E}_+$ **do**
  $\theta_{atmost} = \min(\theta_{atmost}, \texttt{evalSimFn}(e, a, f))$
$\theta_{atleast} \leftarrow 0.0$
for $e \in \mathbf{E}_-$ **do**
  $\theta_{atleast} = \max(\theta_{atleast}, \texttt{evalSimFn}(e, a, f))$
if $\theta_{atleast} < \theta_{atmost}$ **then**
  exists $\leftarrow$ true
  $\theta \leftarrow \frac{\theta_{atleast} + \theta_{atmost}}{2}$
else
  exists $\leftarrow$ false

We will present the SKETCH formulation of the **EM-GBF** problem later in the appendix as implementation notes (appendix A.3). We wrap up this discussion with an end-to-end example below.

An end-to-end EM-GBF example

Here, we put together a sample grammar and constraints to show how to obtain a **GBF** using SKETCH.

Example 4: Consider the example in Figure 1-4. A specific grammar $G^4_{\mathbf{GBF}}$ for representing a Boolean formula matching rule (**GBF**) in this scenario is obtained by using the following in the above definition of the grammar $G_{\mathbf{GBF}}$:

– let $n = 5$ (number of aligned attributes),

– let $\mathcal{F} = \{\texttt{Equal}, \texttt{Levenshtein}, \texttt{Jaccard}\}$,

– let examples be: matching $\mathbf{M} = \{(r_1, s_1), (r_2, s_1)\}$ and non-matching $\mathbf{D} = \{(r_1, s_2)\}$

Our system gives the synthesizer a table representing the evaluations of each similarity function $f \in \mathcal{F}$ on each attribute $i \in [1, n]$ of every provided example $(r, s) \in \mathbf{E}$ (the function `evalSimFn` in the custom synthesizer) e.g.,

$$\begin{aligned}
&\Theta(1, \texttt{Equal}) \\
&\qquad = \{\texttt{Equal}(\text{`Catherine Zeta-Jones'}, \text{`Catherine Zeta-Jones'}), \\
&\qquad\quad \texttt{Equal}(\text{`C. Zeta-Jones'}, \text{`Catherine Zeta-Jones'})\} \\
&\qquad = \{1.0, 0\}
\end{aligned}$$

The **GBF** $\varphi_2 \vee \varphi_3$ from Example 2 can now be obtained as candidate **GBF** from this grammar $G^4_{\mathbf{GBF}}$. □

Chapter 3

Specialization information extraction

Specialization information extraction, as the name suggests, is the identification of specialization information α_i that is used in the synthesis-of-components step (Chapter 2) to synthesize components in the overall framework (Fig. 1-1). The specialization information is used to constrain the synthesis of the components to those that are more likely to improve the numerical score. Moreover, this may lead to more efficient synthesis of components.

In this chapter we elaborate on the kinds of specialization information used in Swapper (Sec. 3.1) and EM-Synth (Sec. 3.3). We also describe the techniques used in Swapper and EM-Synth to find their corresponding specialization information in Sections 3.2 and 3.4, respectively. We start with discussing the specialization information in Swapper below.

3.1 Specialization information in Swapper

To describe the specialization information used in Swapper, we first discuss the components of the simplifier i.e., the conditional rewrite rules (Sec. 2.2). 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. We use the notation $CRR(x) \equiv (LHS(x), pred(x), RHS(x))$ to represent a conditional rewrite rule.

The specification for correctness of a conditional rewrite rule $CRR(x)$ is given by:

$$Spec_{correct}\left(CRR\left(x\right)\right) \equiv \Big(\forall x\ pred\left(x\right)\ \implies\ \left(LHS\left(x\right) = RHS\left(x\right)\right)\Big)$$

The overall specification used for synthesis $Spec_{C_i}(CRR(x))$ contains more constraints than just the correctness of the conditional rewrite rule $CRR(x)$. The motivation behind adding these extra constraints is to synthesize rewrite rules that are not only correct but also applicable at least once in the input corpus of formulas. Formally, the overall specification $Spec_{C_i}(CRR(x))$ can be written as:

$$Spec_{C_i}(CRR(x)) \equiv Spec_{correct}\left(CRR\left(x\right)\right) \wedge Spec_{Score}\left(\alpha_i, CRR\left(x\right)\right)$$

These extra constraints represented by $Spec_{Score}\left(\alpha_i, CRR\left(x\right)\right)$ are derived from the specialization information α_i. The specialization information α_i in SWAPPER has two parts:

1. **A concrete pattern** $Q(x)$. This part of the specialization information is used to constrain the rules to have a fixed LHS pattern and is extracted from the input corpus of formulas.

2. **Contextual assumptions** $assume(x) = \{assume_j(x) | 1 \leqslant j \leqslant m\}$. As described in Subsec. 2.2.1 and Subsec. 2.2.2, SWAPPER collects contextual assumptions for all occurrences of the pattern $Q(x)$ in the input corpus of formulas. In other words, for every occurrence of the pattern $Q(x)$ in the corpus of formulas: (1) SWAPPER identifies the ranges of values for sub-terms corresponding to each of the input variables in $x = (x_1, x_2, ..., x_n)$. These ranges of values can be inferred by the simplifier when applying rewrite rules. (2) SWAPPER builds the predicate $assume_j(x)$ for the j^{th} occurrence of the pattern $Q(x)$ that evaluates to true iff the values for the inputs in x lie in their corresponding ranges of values for this occurrence.

Formally, the specialization information α_i is a tuple $\big(Q(x), assume(x)\big)$, and the overall specification $Spec_{C_i}(CRR(x))$ for $CRR(x) = (LHS(x), pred(x), RHS(x))$ can be written as:

$$\begin{aligned} &Spec_{C_i}\big(\,(LHS(x), pred(x), RHS(x))\,\big) \\ &\quad \equiv \Big(\forall x\ pred(x)\ \implies\ \big(LHS(x) = RHS(x)\big)\Big) \wedge \Big(\forall x\ LHS(x) = Q(x)\Big) \end{aligned}$$

$$\wedge \left(\bigvee_{j=1}^{m} \big(\forall x \ assume_j(x) \implies pred(x) \big) \right)$$

To efficiently synthesize a rewrite rule $CRR(x) = (LHS(x), pred(x), RHS(x))$ we simplify the problem to that of finding a pair $\big(pred(x), RHS(x)\big)$ satisfying the following specification:

$$Spec_{C_i}\big((Q(x), pred(x), RHS(x)) \big)$$
$$\equiv \Big(\forall x \ pred(x) \implies \big(Q(x) = RHS(x) \big) \Big) \wedge \left(\bigvee_{j=1}^{m} \big(\forall x \ assume_j(x) \implies pred(x) \big) \right)$$

A solution $\big(pred(x), RHS(x)\big)$ satisfying the above specification can be combined with $Q(x)$ to find the required rewrite rule $(Q(x), pred(x), RHS(x))$.

As described above, the contextual assumptions $assume(x)$ are extracted from every occurrence of a pattern. But, not all patterns are equally important. Among all the patterns available in the corpus of formulas, the ones that occur more often are more promising for simplification. In the following section 3.2, we describe a representative sampling technique that identifies promising LHS patterns $Q(x)$ for synthesizing conditional rewrite rules.

3.2 Representative sampling of patterns in Swapper

We focus on the problem of identifying promising LHS patterns for synthesizing conditional rewrite rules. The pattern-finding phase in Swapper (Fig. 1-2) takes as input a corpus of formulas $\{P\}$ and uses a representative sampling scheme (explained below) 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.

Note that this problem of finding repeating sub-terms is similar in essence to the motif discovery problem [92], famous for its application in DNA fingerprinting [65]. However, existing techniques used for solving the motif discovery problem are not directly usable in Swapper because they lead to loss of the contextual assumptions required for rule synthesis in Swapper (see Subsec. 9.2.2 for more details).

Probabilistic pattern sampling

A pattern in the context of SWAPPER is an expression tree that has free variables as leaves. Our goal for the pattern-finding phase is to generate a representative sample of patterns S from a corpus of DAGs $\{P\}$ representing input formulas. 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. 3-1, 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. 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.

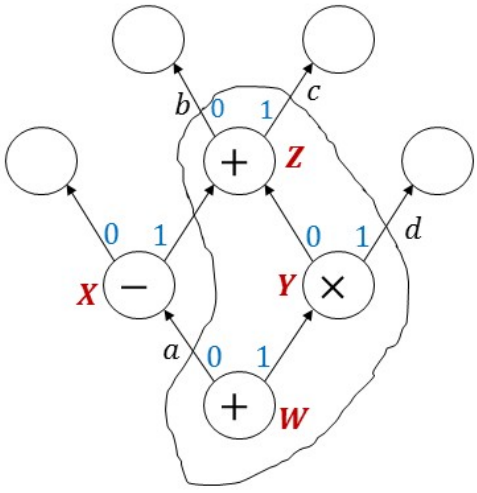

Figure 3-1: Pattern from a rooted sub-graph

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*

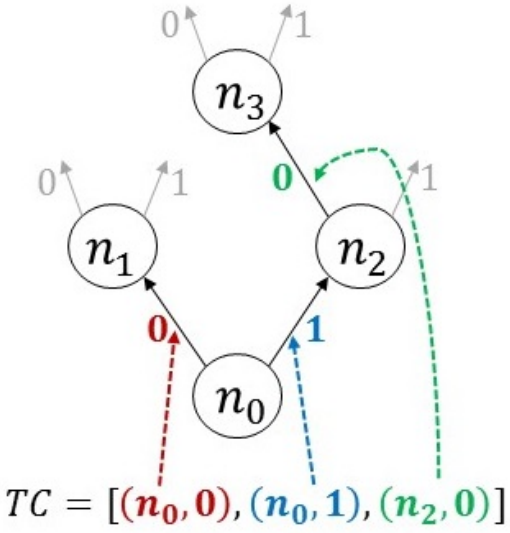

Figure 3-2: Example tree construction

sampling uniformly from the set $\bigcup_i Sub_K(P_i)$*, and then mapping each of the resulting sub-graphs to its 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* $K \geqslant 2$ *and arity* $\delta \geqslant 1$ *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* $i \leqslant K-2$*. We use* $Tree(\tau)$ *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 $Tree(\tau)$ as shown in Fig. 3-2.

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 3: Uniform sampling for TCs

input : $K \geqslant 2$: Size of the TC to be found

$\delta \geqslant 1$: 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

$\tau \leftarrow List()$

$\Lambda \leftarrow List()$ ▷ maintains adjacent/boundary edges

foreach $i \in [0, 1, \ldots, K-2]$ **do**

  foreach $j \in [0, 1, \ldots, \delta - 1]$ **do**

    Λ.append$\big((n_i, j)\big)$ ▷ adds δ edges to boundary

  $(s, t) \leftarrow$ **sample** (Λ) ▷ boundary $|\Lambda| = \big((i+1)\delta\big) - i$

  τ.append$\big((s, t)\big)$

  Λ.remove$\big((s, t)\big)$ ▷ removes an edge from boundary

It is easy to see that any TC of size K will be sampled by Algorithm 3 with a probability of $\Pi_{0 \leqslant i < K-1} \frac{1}{((i+1)\delta) - i}$ because $|\Lambda| = \big((i+1)\delta\big) - i$ at the i^{th} step and we sample uniformly from Λ for each $0 \leqslant i < K-1$. This probability is independent of the TC being considered and hence, every TC is equally likely to be sampled by Algorithm 3.

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 $\bigcup_i \text{nodes}\big(P_i\big) \times \mathbf{TC}$, 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 [72] 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, show that it is bijective and discuss the structure of the final algorithm for representative sampling.

Definition 3 *Canonical set: Given a corpus of formulas* $\{P_i\}$*, we define Canonical* $\subset \bigcup_i nodes\big(P_i\big) \times \mathbf{TC}$ *as the set of tuple-pairs* $\big(\eta, \tau\big)$ *with* $\eta \in nodes\,(P_i)$ *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. 3-1, 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 $\mu((\eta_S, \tau_S)) = S$. 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 $\eta_1 = \eta_2$.

Putting the pieces together

We describe an efficient composite algorithm of rejection sampling, checking BFS order and application of the function μ as described above.

The details of subroutines in Algorithm 4 are as follows:

- **extendBoundary** $(\Lambda, \Psi, \eta.\text{parents})$: This subroutine extends the boundary list Λ and adds all unseen nodes (that aren't already in the Ψ set) from η.parents. It also adds these unseen nodes to Ψ marking them as seen (BFS).

Algorithm 4: Probabilistic Sampling for Pattern Finding

input : $\mathcal{B}$: Set of benchmark formulas as DAGs
$N \geqslant 2$: Size of the pattern to be found
output: A sampled pattern of size N

$\delta \leftarrow$ maximum number of parents of any node in $\mathcal{B}$
while True **do**
  $\eta \leftarrow$ **sample**$\big(\text{nodes}(\mathcal{B})\big)$
  $P \leftarrow \mathit{List}\,(\eta)$ ▷ maintains selected pattern nodes
  $\Lambda \leftarrow \mathit{List}\,()$ ▷ adjacent/boundary nodes in BFS order
  $\Psi \leftarrow \mathit{Set}(\eta)$ ▷ set of already-seen nodes for BFS
  while $|P| < N$ **do**
    extendBoundary$\big(\Lambda, \Psi, \eta.\text{parents}\big)$
    $\eta \leftarrow$ **sampleWithNulls**$\big(\Lambda, |P| \times (\delta - 1) + 1\big)$
    if $\eta = \mathit{NULL}$ **then**
      break ▷ restart sampling a new pattern
    else
      pruneBoundary$\big(\Lambda, \eta\big)$
  if $|P| = N$ **then**
    return $\mathit{Graph}(P)$

- **sampleWithNulls** (Λ, M): This subroutine samples from a list of M nodes obtained by extending Λ by appending NULL nodes to it. Note that since at every iteration, the algorithm moves a node form Λ to P and adds at most δ nodes back to Λ, $|\Lambda|$ can increase at most by $(\delta - 1)$ and hence will never exceed $|P| \times (\delta - 1) + 1$.

- **pruneBoundary** (Λ, η): This subroutine removes η and all nodes that occur before the node η from Λ. Note that nodes are added to Λ in the BFS order and hence, this ensures that the nodes are sampled in the increasing BFS order as well.

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. **Contextual assumptions from the benchmark formulas:** The range of values

inferred by the solver (Subsec. 2.2.1 and Sec. 3.1) 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 since the same pattern can occur with different ranges of values, these ranges 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 (e.g., it may stop at $M = 50,000$ and $\epsilon = 0.02$). SWAPPER samples patterns of sizes $2, 3, 4, ...$ and stops after sampling patterns of size 7.

With this pattern-finding technique, SWAPPER is able to find promising LHS patterns and their contextual assumptions that are used in the rule-synthesis step (Sec. 2.2). This wraps up the discussion of specialization-information extraction in SWAPPER. Now, we move to discussing how we find the specialization information in the EM-Synth system that can be used in the EM rule-synthesis step (Sec. 2.3). We describe the specialization information in the context of EM-Synth in Sec. 3.3 and discuss the techniques to iteratively choose different specialization information for the EM rule-synthesis step in Sec. 3.4.

3.3 Specialization information in EM-Synth

To discuss the specialization information in EM-Synth, we first recall that the components in EM-Synth are entity matching (EM) rules that have a Boolean structure around some atomic predicates (Sec. 2.3). We also recall that the user provides a set of examples, denoted by $\mathbf{E} = \mathbf{M} \cup \mathbf{D}$, where $\mathbf{M}$ are positive examples, i.e., pairs of records that represent the same entity, and $\mathbf{D}$ are negative examples, i.e., pairs of records that represent different entities (Sec. 1.2). Each EM rule φ is also required to be from a bounded grammar $G_{\mathbf{GBF}}(N_a, N_d)$ (Subsec. 2.3.2).

The synthesis specification $Spec_{C_i}(\varphi)$ further constrains the synthesis of the component

φ, where

$$Spec_{C_i}(\varphi) \equiv Spec_{correct}(\varphi) \wedge Spec_{Score}(\alpha_i, \varphi)$$

with α_i being the specialization information. In EM-Synth, there are no logical correctness constraints on an EM rule φ i.e., $Spec_{correct}(\varphi) \equiv \mathsf{true}$. So, in principle we can synthesize all EM rules in $G_{\mathbf{GBF}}(N_a, N_d)$ without any other constraints, but it would not be efficient to do so, especially, for larger values of the bounds N_a and N_d. So, the specialization information α_i and the corresponding specification $Spec_{Score}(\alpha_i, \varphi)$ are very important for constraining the synthesis further. The specialization information in EM-Synth is a subset of the user-provided examples. Intuitively, we want to ensure that the synthesized rules correctly match as many of the user-provided examples as possible. We specialize the overall synthesis specification $Spec_{C_i}(\varphi)$ by constraining φ to match some of the user-provided examples correctly i.e.,

$$Spec_{C_i}(\varphi) \equiv Spec_{Score}(\alpha_i, \varphi) \equiv \bigwedge_{(r,s)\in\mathbf{M}_{\mathrm{SYN}}} \varphi(r,s) \wedge \bigwedge_{(r,s)\in\mathbf{D}_{\mathrm{SYN}}} \neg\varphi(r,s)$$

where $\mathbf{M}_{\mathrm{SYN}} \subseteq \mathbf{M}$ and $\mathbf{D}_{\mathrm{SYN}} \subseteq \mathbf{D}$. This specification is used to synthesize the EM rules from a small set of examples (as described in Sec. 2.3). The specialization information α_i, which is used to build the specification $Spec_{Score}(\alpha_i, \varphi)$, is a subset of examples $\alpha_i = \mathbf{M}_{\mathrm{SYN}} \cup \mathbf{D}_{\mathrm{SYN}}$ chosen from the input set of examples $\mathbf{M}_{\mathrm{SYN}} \subseteq \mathbf{M}$ and $\mathbf{D}_{\mathrm{SYN}} \subseteq \mathbf{D}$.

In the next section (Sec. 3.4), we discuss the ideas and techniques used to iteratively find the specialization information i.e., the subsets of examples to use for EM rule synthesis.

3.4 Choosing sets of examples in EM-Synth

Specialization-information extraction in EM-Synth corresponds to identifying subsets of examples that are used to synthesize corresponding EM rules or **GBF**s (Sec. 2.3) in an iterative loop (Fig. 1-5). When choosing the examples for synthesizing a **GBF**, we want the **GBF**s to match as many examples as possible correctly. But, there are some *inconsistent* examples that we want to avoid as well. These inconsistent examples (1) can be incorrect labels on the examples provided by the user or (2) may require some external information for matching them correctly that is not captured by the similarity functions used in EM-Synth e.g., matching the names "Robert Baratheon" and "Bob B." might not be possible without

incorporating the knowledge that "Bob" is a nickname for "Robert" and last names can be represented as initials. Moreover, for some of these inconsistent examples it might be possible to match them in EM-Synth, but that may lead to a **GBF** that does not match many other examples.

Based on the discussion above, we summarize the two goals of specialization-information extraction in EM-Synth:

1. We must choose subsets of **E** (user-provided examples) in a way that allows us to synthesize a **GBF** with good coverage of the examples i.e., we would want the synthesized **GBF** to be likely to correctly match many different kinds of examples.

2. We have to avoid *inconsistent* examples that either lead to no **GBF** or a **GBF** that does not match many other examples.

Keeping these goals in mind, we introduce our approach to specialization-information extraction in EM-Synth, designed to satisfy the above goals.

For Goal 1., we use ideas from Counter-Example Guided Inductive Synthesis (CEGIS) [112] to perform synthesis from a few examples – described in detail below in Subsec. 3.4.1.

For Goal 2., we are inspired by Random Sample Consensus (RANSAC) [49] to avoid inconsistent examples – described in Subsec. 3.4.2.

3.4.1 Synthesis from a few EM examples (CEGIS)

We use ideas from the Counter-Example Guided Inductive Synthesis (CEGIS) [112] approach to build an iterative algorithm referred to as RS-CEGIS (Algorithm 5) that has two phases: **Synth** (line 6) and **Verify** (line 9). The **Synth** phase corresponds to the rule-synthesis step (Sec. 2.3) in EM-Synth. The idea is to iteratively synthesize a **GBF** that works for a small set of examples and expand this set in a smart manner by adding an example that is currently not being handled correctly by the synthesized **GBF**.

The full version of the RS-CEGIS algorithm is presented in Algorithm 5. It has a CEGIS loop (lines 5-16) that picks a random sample of one example $\mathbf{E}_{\text{SYN}} = List(e_0)$ to bootstrap the synthesis algorithm. In each iteration, given a sample $\mathbf{E}_{\text{SYN}}$, it starts with the **Synth** routine (line 6) to synthesize a **GBF** φ_i where i is the CEGIS iteration counter. If it cannot find a satisfiable **GBF**, it will break out of the loop (lines 7-8); otherwise, it will append the

GBF φ_i to the output list $\mathcal{L}_\varphi$ and then **Verify** to find counter-examples (line 10). Either there is no counter-example so the process will terminate by returning only the **GBF** that matches every example correctly (lines 11-12), or a randomly selected counter-example will be added to be considered in the next CEGIS iteration (lines 13-15). Finally, the algorithm will return the list of all **GBF**s (line 17) synthesized across all iterations.

Algorithm 5: CEGIS-based specialization-information extraction (RS-CEGIS)

input : $\mathbf{E} = \mathbf{M} \cup \mathbf{D}$: Set of examples
$G_{\mathbf{GBF}}(N_a, N_d)$: Bounded **GBF** grammar
$\mathcal{F}$: Library of Similarity Functions
K_{CEGIS} : Bound on CEGIS iterations
output: $\mathcal{L}_\varphi$: A list of **GBF**s from $G_{\mathbf{GBF}}(N_a, N_d)$

```
1  L_φ ← List()
2  i ← 0
3  e_0 ← sample(E)
4  E_SYN ← List(e_0)
5  while i < K_CEGIS do                                   // CEGIS loop
6      φ_i ← Synth(G_GBF(N_a, N_d), E_SYN, F)
7      if φ_i = null then                                 // Unsatisfiable Synth
8          break                                          // restart CEGIS
9      L_φ ← L_φ.append(φ_i)
10     Ē_φi ← Verify(φ_i, E)                              // Counter-examples
11     if Ē_φi = ∅ then
12         return List(φ_i)                               // φ_i Works on all examples!
13     else
14         e_{i+1} ← sample(Ē_φi)
15         E_SYN ← E_SYN.append(e_{i+1})
16     i ← i + 1
17 return L_φ
```

Note that, at iteration i, **Synth** uses the currently available examples $\mathbf{E}_{\text{SYN}} = \{e_0, e_1, \ldots, e_i\}$ and solves the exact **EM-GBF** rule-synthesis problem with SKETCH to find a **GBF** φ_i from the bounded grammar $G_{\mathbf{GBF}}(N_a, N_d)$ that correctly handles all the examples in $\mathbf{E}_{\text{SYN}}$ (as described in Sec. 2.3). **Verify**, on the other hand, considers the full set of examples $\mathbf{E} = \mathbf{M} \cup \mathbf{D}$ and finds the *counter-example* subset $\overline{\mathbf{E}}_{\varphi_i} \subset \mathbf{E}$, which contains examples $e \in \mathbf{E}$ such that $\varphi_i(e) = \mathsf{false}$ if $e \in \mathbf{M}$ and $\varphi_i(e) = \mathsf{true}$ if $e \in \mathbf{D}$. In other words, it identifies examples that are incorrectly handled by φ_i. A counter-example e_{i+1} chosen randomly from $\overline{\mathbf{E}}_{\varphi_i}$ is added to the set $\mathbf{E}_{\text{SYN}}$ to be considered in the next **Synth** phase. The process continues until either **Synth** is unable to find a **GBF** for the current set of

examples or until it has performed K_{CEGIS} (*CEGIS cutoff*) iterations. If **Verify** cannot find any counter-example (i.e., $\overline{\mathbf{E}}_{\varphi_i} = \varnothing$), the algorithm terminates and outputs $List(\varphi_i)$ as the optimal list of one **GBF** since it correctly handles all examples in **E**.

For example, consider Figure 1-4 (from Sec. 1.2) with matching examples $\mathbf{M} = \{(r_1, s_1), (r_4, s_2), (r_2, s_1)\}$ and non-matching examples $\mathbf{D} = \{(r_1, s_2), (r_4, s_1)\}$. Suppose the algorithm picks (r_1, s_1) as the first example and **Synth** returns the function $\varphi_0 =$ `Equal[name]` $\geqslant 1.0$. **Verify** tries this function on all examples in $\mathbf{M} \cup \mathbf{D}$ and randomly picks (r_4, s_1) as the counter-example, i.e., an example which is not correctly matched by the function φ_0 since the names are not equal for (r_4, s_1). It would then add this counter-example to the set $\mathbf{E}_{\text{SYN}}$ and start the next CEGIS iteration. In this iteration **Synth** may now return the function $\varphi_1 =$ `Jaccard[name]` $\geqslant 0.4$, which matches all examples correctly.

In summary, the RS-CEGIS algorithm uses the SKETCH solver multiple times on smartly chosen sets of examples to find EM rules that cover many examples. Now, we address the second goal from above and describe how we use RANSAC to avoid inconsistent examples.

3.4.2 Synthesis with inconsistent examples (RANSAC)

We use ideas from the Random Sample Consensus (RANSAC) [49] approach and build a loop (described in Algorithm 6) on top of the RS-CEGIS algorithm to restart it multiple times with different initial random examples (e_0). The idea is that if the provided example set contains a small number of inconsistent examples, then multiple runs are more likely to avoid them. Note that some examples individually may not be inconsistent, i.e., the algorithm may still find a good **GBF** after choosing them in the CEGIS loop. Instead, certain larger subsets of examples may correspond to conflicting constraints on the **GBF** grammar and constitute inconsistency only when all examples in that subset are chosen together. Both the randomness in the **sample** routine (Algorithm 5) and the RANSAC restarts help avoid choosing all such examples together. Before restarting RS-CEGIS, if the number of restarts reaches K_{RANSAC} (the RANSAC cutoff) then the algorithm terminates and outputs the list of all **GBF**s $\mathcal{L}_\varphi$ seen across all CEGIS and RANSAC iterations.

The RS-RANSAC algorithm, combining ideas from both CEGIS and RANSAC, is presented in Algorithm 6.The RS-RANSAC algorithm has an outer (RANSAC) loop (lines 3-6) that picks random samples to bootstrap the synthesis algorithm.In each iteration, it invokes the inner RS-CEGIS algorithm at line 4 which corresponds to the RS-CEGIS algorithm

Algorithm 6: RS-RANSAC algorithm for specialization information extraction in EM-Synth

input : $\mathbf{E} = \mathbf{M} \cup \mathbf{D}$: Set of examples
$G_{\mathbf{GBF}}(N_a, N_d)$: Bounded **GBF** grammar
$\mathcal{F}$: Library of Similarity Functions
K_{RANSAC} : Bound on RANSAC restarts
K_{CEGIS} : Bound on CEGIS iterations

output: $\mathcal{L}_\varphi$: A list of **GBF**s from $G_{\mathbf{GBF}}(N_a, N_d)$

1 $r \leftarrow 0$
2 $\mathcal{L}_\varphi \leftarrow List()$
3 **while** $r < K_{RANSAC}$ **do** `// RANSAC loop`
4 $\quad \mathcal{L}'_\varphi \leftarrow \text{RS-CEGIS}\left(\mathbf{E}, G_{\mathbf{GBF}}\left(N_a, N_d\right), \mathcal{F}, K_{\text{CEGIS}}\right)$ `// from Algorithm 5`
5 $\quad \mathcal{L}_\varphi \leftarrow \mathcal{L}_\varphi.\mathbf{extend}(\mathcal{L}'_\varphi)$
6 $\quad r \leftarrow r + 1$
7 **return** $\mathcal{L}_\varphi$

(Algorithm 5).The RS-CEGIS algorithm is explained in Subsec. 3.4.1. It then adds the synthesized **GBF**s to the list of **GBF**s $\mathcal{L}_\varphi$ (line 5) and eventually outputs the list after at most K_{RANSAC} iterations.

Chapter 4

Assembly of components

Once we have the synthesized components or rules, we have to combine them to the required output function in a way that will guarantee the correctness of the function. More formally, given a set of synthesized components $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$, we need a procedure $Assemble(\pi, \mathcal{C})$ such that the following property holds:

$$\forall_{\pi \in \Pi} \Bigg(\Big(f = Assemble(\pi, \mathcal{C}) \wedge \bigwedge_{i=1}^{n} \big(C_i \in G_C \wedge Spec_{C_i}(C_i) \big) \Big) \implies \Big(f \in G \wedge \forall_{in \in A}\ Spec(f, in) \Big) \Bigg)$$

where $C_i \in G_C \wedge Spec_{C_i}(C_i)$ is the synthesis specification for the component C_i and $Spec(f, in)$ is the correctness specification for the output function f. Assuming that the components satisfy their synthesis specifications (since they were synthesized from these specifications, they must satisfy them), we can rewrite the required property as:

$$\forall_{\pi \in \Pi} \Big(f = Assemble(\pi, \mathcal{C}) \implies \big(f \in G \wedge \forall_{in \in A}\ Spec(f, in) \big) \Big)$$

In this chapter, we describe the assembly procedure *Assemble* and the parameter space Π used in each of the SWAPPER and EM-Synth systems. We also argue that the above property (referred to as the *soundness* of assembly) holds in both of these systems. We start with a discussion of the assembly procedure, its parameter space and soundness in SWAPPER (Sec. 4.1). We also discuss the techniques enabling the assembly in SWAPPER i.e., rule generalization (Subsec. 4.1.2) and pattern matching (Subsec. 4.1.3). In Sec. 4.2,

we discuss the two studied assembly procedures, their parameter spaces and their soundness in EM-Synth. We further discuss the details of the two studied assembly procedures i.e., Boolean combination (in Subsec. 4.2.1) and consensus building (in Subsec. 4.2.2).

4.1 Assembly in SWAPPER

The assembly procedure in SWAPPER generates a simplifier f from a set of rewrite rules (components) $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$. The assembled simplifier takes as input a formula P and returns a simplified formula P' and has a lot of flexibility in terms of:

- How to maintain and modify the formula while rewriting?
- How to apply the rewrite rules?
- Which rules to apply?
- What order to apply them in?

In SWAPPER, the assembled simplifier uses the existing SKETCH solver infrastructure to perform simplification and fixes some of the flexible choices mentioned above. In particular, the simplifier works as follows:

1. Formulas are represented and maintained as topologically sorted DAGs.
2. Simplification is done in one forward pass of the topologically sorted DAG. All rules are pattern matched and applied (if possible) at each node.
3. When applying a particular rewrite rule $LHS(x) \xrightarrow{pred(x)} RHS(x)$, no nodes are immediately removed. Instead, a copy of all nodes of $RHS(x)$ as a topologically sorted DAG is added to the existing formula DAG. The input and output connections of the $LHS(x)$ pattern are accordingly moved to the $RHS(x)$ DAG. Each new node in the copy of $RHS(x)$ is simplified using the same steps (items 2., and 3., of this list) before being added to the formula DAG.
4. At the end, the unreachable patterns are removed from the DAG i.e., any node whose output is not being used by another node is removed from the DAG, and this process is continued recursively until no such nodes can be removed.

There are still some choices (parameters π) that the assembly procedure needs to make in the above description. The parameters of this assembly procedure in SWAPPER are the following:

- Subset of rewrite rules $\mathcal{C}_f \subseteq \mathcal{C}$ to use in the simplifier.
- An ordering of the rewrite rules in $\mathcal{C}_f$ to use while checking for application of the rules at a node.

More formally, the parameter space Π of the assembly procedure in SWAPPER is given by:

$$\Pi = \Big\{ (\mathcal{C}_f, \sigma_f) \mid \mathcal{C}_f \subseteq \mathcal{C} \ \wedge \ \big(\sigma_f \text{ is a permutation of the set } \mathcal{C}_f\big) \Big\}$$

For n components the size of the parameter space can be estimated as follows:

$$|\Pi| = \sum_{k=0}^{n} \binom{n}{k} \times k! = \sum_{k=0}^{n} \frac{n!}{(n-k)!} = n! \sum_{k=0}^{n} \frac{1}{k!} \approx e \ n!$$

where $e \approx 2.71828$ is Euler's number. One can see that the number of possible parameters in SWAPPER grows as quickly as the factorial function with the value of n, and it becomes really hard to enumerate all the parameters efficiently e.g., the number of possible parameters for $n = 200$ (number of rules generated for a domain considered in our experiments) is approximately 10^{375}.

Now, we briefly discuss the soundness of assembly in SWAPPER.

4.1.1 Soundness of assembly

For a simplifier f, the correctness of f corresponds to f maintaining the semantics of the formula being simplified. For an input formula P, the simplifier f will make changes to P only when applying a conditional rewrite rule locally. A typical transformation when applying a rewrite rule $LHS(x) \xrightarrow{pred(x)} RHS(x)$ in a formula $P(y)$ (with inputs y) can be represented as:

$$P_1\Big(b_1(y), \ldots, LHS\big(a(y)\big), \ldots, b_m(y)\Big) \xrightarrow{pred(a(y))} P_1\Big(b_1(y), \ldots, RHS\big(a(y)\big), \ldots, b_m(y)\Big)$$

where the formula $P(y)$ is written as another formula P_1 with m inputs derived from its original input y, and one of those m inputs corresponds to an occurrence of the $LHS(x)$

pattern. The inputs x to the LHS match with the outputs of another function $a(y)$. This transformation of the formula $P(y)$ can only be done when the simplifier can prove that $pred(a(y))$ is true for any value taken by y. Assuming correctness of the rewrite rule, we have

$$\forall x \left(pred(x) \implies \big(LHS(x) = RHS(x)\big) \right)$$

The above constraint must hold for $x = a(y)$ for any y and since $pred(a(y)) = \mathsf{true}$, we have

$$\forall y \; LHS(a(y)) = RHS(a(y))$$

Hence, we conclude that

$$\forall y \; P_1\Big(b_1(y), \ldots, LHS\big(a(y)\big), \ldots, b_m(y)\Big) = P_1\Big(b_1(y), \ldots, RHS\big(a(y)\big), \ldots, b_m(y)\Big)$$

This argument shows that a simplifier of formulas based on local rewriting will be correct (i.e., semantics preserving) if the local rewrite rules used in the simplifier are correct themselves. Note that all we are showing here is that if the simplifier performs a series of rewrites then the rewritten formula has the same semantics as the input formula. It may be possible that this simplifier never terminates. The soundness property of assembly is still valid in that case.

Also, note that this argument works irrespective of the choices of the parameters (which rules to apply and in what order). This argument can be applied to the DAG-based rewriting in SKETCH as well, where DAGs enable sharing between different parts of the formula P, thereby, allowing rewrites using one rule at multiple places (in the abstract representation of the formula) where the LHS pattern occurs in the formula (as presented above) at the same time.

Termination of simplifiers assembled in SWAPPER

As discussed above, general simplifiers based on rewrite rules may never terminate [15, 55] e.g., a rewrite rule may replace a term $A(x)$ with $B(x)$ and then another rewrite rule may replace $B(x)$ with $A(x)$. Even in the context of SWAPPER, the generated simplifiers may not always terminate. For example, consider the simplifier that applies the following two

rules:

$$(1)\quad and(neg(x), neg(y)) \xrightarrow{\text{true}} neg(or(x, y))$$

$$(2)\quad mux(z, neg(x), neg(y), neg(or(x, y))) \xrightarrow{\text{true}} mux(z, neg(x), neg(y), and(neg(x), neg(y)))$$

Note that in the rules above: x, y are Boolean input variables and z is an integer variable. Also, even though the RHS of rule (2) looks larger than the LHS, due to sharing of nodes in its DAG representation, the DAG size of the RHS is only 4 whereas the DAG size of LHS is 5. On an input DAG $mux(z, neg(x), neg(y), neg(or(x, y)))$ to the simplifier, it will result into a non-terminating cycle of rewrites:

$$\begin{aligned} &mux(z, neg(x), neg(y), neg(or(x, y))) \\ &\xrightarrow{\text{rule (2)}} mux(z, neg(x), neg(y), and(neg(x), neg(y))) \\ &\xrightarrow{\text{rule (1)}} mux(z, neg(x), neg(y), neg(or(x, y))) \\ &\xrightarrow{\text{rule (2)}} \ldots \end{aligned}$$

SWAPPER can eliminate non-terminating simplifiers by discarding simplifiers that take more time than a particular timeout during the auto-tuning step (Sec. 5.1). This timeout is set based on the existing simplifier in SKETCH. In our experiments, all simplifiers assembled by SWAPPER were terminating on all input DAGs provided to them.

Now, we are ready to discuss techniques that are used to assemble an efficient and effective simplifier in SWAPPER. We discuss rule generalization in Subsec. 4.1.2 and pattern matching in Subsec. 4.1.3.

4.1.2 Generalization of rewrite rules

One important optimization while assembling a simplifier in SWAPPER is *rule generalization*. The goal of rule generalization is to make the rule more applicable by eliminating redundant nodes from the LHS and RHS patterns. 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:

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

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 below:

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

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.

Given a conditional rewrite rule $LHS(x) \xrightarrow{pred(x)} RHS(x)$, we present the list of rule-generalization strategies employed in SWAPPER in the following subsections. Before delving into the details, we clarify the notation used below and the context in which these generalization strategies are applied:

1. We use $\equiv$ to denote symbolic equivalence of two expressions or patterns in the internal language of SKETCH constraints described in Fig. 2-3

2. SWAPPER stores the patterns $LHS(x)$ and $RHS(x)$ as directed acyclic graphs (DAGs) for sharing sub-patterns and efficient implementation of these strategies

3. A node in the DAG representing a pattern in this language is represented as $op\,(e_0, e_1, \ldots, e_j)$ where op is an operation (as described in Fig. 2-3) and $e_0, e_1, \ldots, e_j$ are also patterns in the same language.

4. The strategies described below are recursively applied in SWAPPER until no more generalizations can be done

Now we are ready to discuss the aforementioned strategies used for rule generalization in SWAPPER.

Common sub-pattern removal from top

SWAPPER finds a node (or sub-pattern) $n_l \equiv op_l(e_0, e_1, ..., e_j)$ (corresponding to an operation op_l like *plus* from Fig. 2-3) in $LHS(x)$ and a node $n_r \equiv op_r(f_0, f_1, ..., f_k)$ in $RHS(x)$ such that $op_l \equiv op_r, j = k$ (they both correspond to the same operation with the same number of arguments) and $\forall\ i \in [0, j] : e_i \equiv f_i \wedge e_i \in \texttt{set}(x)$ i.e., the arguments of both operations are the same input variables from the input vector $x = (x_1, x_2, ..., x_n)$. For such a pair of nodes (n_l, n_r): SWAPPER replaces the nodes with a new input variable x_{n+1} in $LHS(x)$ and $RHS(x)$ patterns and constructs a new rewrite rule:

$$LHS(x')[n_l \rightarrow x_{n+1}] \xrightarrow{pred(x')} RHS(x')[n_r \rightarrow x_{n+1}]$$

where $x' = (x_1, x_2, ..., x_n, x_{n+1}),\ pred(x') \equiv pred(x)$. Clearly, this new rule is more applicable than the original rule, but it may not be a correct rule. For example, if we consider the following rule:

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

the sub-pattern $lt(a, d)$ cannot be replaced with a new input variable e because the new rule:

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

would be incorrect. One may be tempted to assume that it is okay to do such transformations when the inputs involved in the sub-pattern are not used in the predicate $pred(x)$, but that is still not sufficient to guarantee correctness of the new rule, as shown by the rule below:

$$mod(mod(a, b), b) \xrightarrow{true} mod(a, b)$$

the sub-pattern $mod(a, b)$ cannot be replaced with a new variable c because the rule

$$mod(c, b) \xrightarrow{true} c$$

is incorrect.

For this reason, SWAPPER fully verifies the new rule using an off-the-shelf SMT solver named `Z3` [41], and if the rule is correct it keeps the new generalized rule and removes the older rule. Otherwise, it discards the generalized rule. Note that the requirement for each

argument of an operation matching precisely (in the order these arguments are specified) can be loosened for commutative operations like *plus*, *times*, etc, so that, even if SWAPPER finds $plus(x_1, x_2)$ in $LHS(x)$ and $plus(x_2, x_1)$ in $RHS(x)$, it will still try to perform this generalization.

An example application of this generalization strategy is given below.

Initial Rule:

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

Generalized Rule:

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

Common sub-pattern removal from bottom

Assume that the patterns $LHS(x)$ and $RHS(x)$ have op_l and op_r as their root (or base) operations i.e., $LHS(x) \equiv op_l(e_0, e_1, ..., e_j)$ and $RHS(x) \equiv op_r(f_0, f_1, ..., f_k)$. If $op_l \equiv op_r, j = k$ and $\exists i^* \in [0, j]$ such that $\forall\ i \in [0, i^*) \cup (i^*, j] : e_i \equiv f_i$ i.e., all except one of the parents of the base operation match syntactically. In this case, SWAPPER constructs the rule

$$e_{i^*}(x) \xrightarrow{pred(x)} f_{i^*}(x)$$

and verifies the new rule using `z3` [41]. If the rule is correct it keeps the new generalized rule and removes the older rule. Otherwise, it discards the generalized rule. Similar to the previous generalization, the requirement for each argument of the operation op_l matching precisely (in order) can be loosened when it is a commutative operation like *plus*, *times*, etc.

An example application of this generalization strategy is given below.

Initial Rule:

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

Generalized Rule:

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

Replacing nodes with new inputs recursively from top in the LHS

For every node $n_l \equiv op_l(e_0, e_1, ..., e_j)$ in $LHS(x)$ such that each of its parents is an input variable from $x = (x_1, x_2, ..., x_n)$ i.e., $\forall\ i \in [0, j] : e_i \in \texttt{set}(x)$, SWAPPER replaces the node n_l with a new input variable x_{n+1} in $LHS(x)$ to generate a new rule:

$$LHS(x')[n_l \rightarrow x_{n+1}] \xrightarrow{pred(x')} RHS(x')$$

where $x' = (x_1, x_2, ..., x_n, x_{n+1})$, $pred(x') \equiv pred(x)$ and $RHS(x') \equiv RHS(x)$. SWAPPER then fully verifies the new rule using `z3` [41]. If the rule is correct it keeps the new generalized rule and removes the older rule. Otherwise, it discards the generalized rule.

An example application of this generalization strategy is given below.

Initial Rule:

$$times(plus(a, b), x) \xrightarrow{x==0} n(0)$$

where $n(c)$ denotes a constant integer value, $c = 0$ in this case.

Generalized Rule:

$$times(y, x) \xrightarrow{x==0} n(0)$$

Generating all equivalent rules modulo commutativity

During this phase in SWAPPER, all rules with commutative operations in $LHS(x)$ are duplicated with semantically equivalent but syntactically different LHS patterns generated by choosing different ordering of arguments for each commutative operation in LHS. For example, the following rule will result into multiple rules as listed below:

Initial rule:

$$and(lt(plus(a, b), c), lt(plus(b, d), c)) \xrightarrow{d<a} lt(plus(a, b), c)$$

Generalized rules (8 rules including the original one because of 3 commutative operations: 2 *plus*'s and 1 *and*):

Changing ordering of arguments for the two *plus*'s

$$and(lt(plus(a, b), c), lt(plus(b, d), c)) \xrightarrow{d<a} lt(plus(a, b), c)$$

$$and(lt(plus(b,a),c),lt(plus(b,d),c)) \xrightarrow{d<a} lt(plus(a,b),c)$$

$$and(lt(plus(a,b),c),lt(plus(d,b),c)) \xrightarrow{d<a} lt(plus(a,b),c)$$

$$and(lt(plus(b,a),c),lt(plus(d,b),c)) \xrightarrow{d<a} lt(plus(a,b),c)$$

Changing ordering of the arguments of *and* and then again, changing ordering of arguments for the two *plus*'s

$$and(lt(plus(b,d),c),lt(plus(a,b),c)) \xrightarrow{d<a} lt(plus(a,b),c)$$

$$and(lt(plus(b,d),c),lt(plus(b,a),c)) \xrightarrow{d<a} lt(plus(a,b),c)$$

$$and(lt(plus(d,b),c),lt(plus(a,b),c)) \xrightarrow{d<a} lt(plus(a,b),c)$$

$$and(lt(plus(d,b),c),lt(plus(b,a),c)) \xrightarrow{d<a} lt(plus(a,b),c)$$

In this subsection, we described the strategies used to generalize the rules synthesized in SWAPPER. These strategies enabled SWAPPER to (1) discard rules that are same as or weaker than a more applicable generalized rule and (2) reduce the size of the search space for auto-tuning the set and ordering of rules (to be discussed in Sec. 5.1). In the next subsection, we briefly discuss how we generate efficient code for pattern matching and applying the synthesized rewrite rules.

4.1.3 LALR-style pattern matching in SWAPPER

SWAPPER reduces 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 [86]. SWAPPER outputs efficient C++ code that (1) performs LALR-style pattern matching on the provided LHSs and (2) does local formula rewriting based on the given rules (in a particular order of application). The details of the pattern-matching algorithm and choices of the data structures made are explained in Chapter 6 of my master's thesis [103]. Note that SWAPPER is able to do such optimizations automatically because the code of the simplifier doesn't have to be readable or maintainable. Instead of adding code for a new rule, a developer may add the abstract rule to the list of rules generated by SWAPPER and press a button to compile all the rules

together into efficient C++ code.

This section concludes our discussion of assembly in SWAPPER. Now, focus on the assembly procedure in EM-Synth and discuss how it may combine multiple EM rules together.

4.2 Assembly in EM-Synth

The assembly of multiple EM rules or **GBF**s is an integral part of the overall synthesis process in EM-Synth (Fig. 1-5). The assembly procedure in EM-Synth generates a composite EM rule f from a set of EM rules (components) $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$ where each component $C_i \in G_{\mathbf{GBF}}$. Since there are no correctness constraints on f i.e., $Spec(f, in) \equiv \mathsf{true}$, the soundness of assembly an be simplified to:

$$\forall_{\pi \in \Pi} \Big(f = Assemble(\pi, \mathcal{C}) \implies f \in G_{\mathbf{GBF}} \Big)$$

In the following two subsections, we discuss two assembly procedures and their corresponding parameter spaces: (1) Boolean combinations (Subsec. 4.2.1) and (2) consensus building (Subsec. 4.2.2).

4.2.1 Boolean combinations of EM rules

This assembly procedure produces larger **GBF**s (or EM rules) from smaller **GBF**s. The larger **GBF**s are a Boolean combination of the existing **GBF**s. More formally, a Boolean combination of **GBF**s from a set $\mathcal{C}_\varphi = \{\varphi_1, \varphi_2, \ldots, \varphi_n\}$ corresponds to a candidate derived from the following grammar G_{COMB}:

$$\begin{aligned}
&\mathsf{grammar} \quad G_{\mathsf{atom}} \rightarrow \varphi \qquad (\varphi \in \mathcal{C}_\varphi) \\
&\mathsf{grammar} \quad G_{\text{COMB}} \rightarrow G_{\mathsf{atom}} \ (\mathbf{bound} : N_a) \\
&\qquad\qquad \left.\begin{aligned} G_{\text{COMB}} &\rightarrow \neg G_{\text{COMB}} \\ G_{\text{COMB}} &\rightarrow G_{\text{COMB}} \wedge G_{\text{COMB}} \\ G_{\text{COMB}} &\rightarrow G_{\text{COMB}} \vee G_{\text{COMB}} \end{aligned}\right\} (\mathbf{depth} : N_d)
\end{aligned}$$

Note that any candidate from the grammar G_{COMB} is also a **GBF** because the production rules of G_{COMB} are the same as that of $G_{\mathbf{GBF}}$ except the terminal variables which are themselves **GBF**s. This establishes the soundness of this assembly procedure i.e., irrespective of the sequence of production rules applied from G_{COMB}, the EM rules in $\mathcal{C}_\varphi$ are

combined to generate an EM rule f which is also a **GBF**.

There are many different ways of combining the smaller **GBF**s in $\mathcal{C}_\varphi$ together using the grammar G_{COMB}. In fact, many of the Boolean combinations derived from G_{COMB} are equivalent e.g., for $\mathcal{C}_\varphi = \{\varphi_1, \varphi_2, \varphi_3\}$, both the formulas $(\varphi_1 \vee \varphi_2) \wedge \varphi_3$ and $(\varphi_1 \wedge \varphi_3) \vee (\varphi_2 \wedge \varphi_3)$ are logically equivalent but correspond to different derivations from G_{COMB}. To avoid deriving logically equivalent formulas, we use a canonical representation of a Boolean formula, i.e. a truth table, and define the parameters in terms of the outputs in the truth table.

Consider the truth table shown in Fig. 4-1. The table represents multiple Boolean combinations of **GBF**s in $\mathcal{C}_\varphi = \{\varphi_1, \varphi_2, \varphi_3\}$ that are logically equivalent to $(\varphi_1 \vee \varphi_2) \wedge \varphi_3$. A different truth table will have different values in the output column. In general, a truth table $\tau : 2^{\mathcal{C}_\varphi} \rightarrow \{\mathsf{true}, \mathsf{false}\}$ maps every subset of the EM rules in $\mathcal{C}_\varphi$ to true or false. If there are n rules in $\mathcal{C}_\varphi$ then there are 2^n rows in the truth table of a Boolean combination of **GBF**s from $\mathcal{C}_\varphi$ and, hence, there are 2^{2^n} possible truth tables. It is easy to transform a truth table to a **GBF**. We use the Quine-McCluskey algorithm [79] to transform a truth table τ to a Boolean formula with **GBF**s from $\mathcal{C}_\varphi$ as its atoms.

φ_1	φ_2	φ_3	*output*
true	true	true	true
true	true	false	false
true	false	true	true
true	false	false	false
false	true	true	true
false	true	false	false
false	false	true	false
false	false	false	false

Figure 4-1: An example truth table for $\mathcal{C}_\varphi = \{\varphi_1, \varphi_2, \varphi_3\}$

Note that it will not be efficient to build truth tables for a large value of $n = |\mathcal{C}_\varphi|$. So, instead we first identify a small number (K) of rules from $\mathcal{C}_\varphi$ and then build truth tables for B out of those K rules at a time. Hence, for given hyper-parameters $\mathcal{C}_K \subseteq \mathcal{C}_\varphi, B$ where $|\mathcal{C}_K| = K$ and $1 \leqslant B \leqslant K$, the parameter space $\Pi(\mathcal{C}_K, B)$ for this assembly procedure is given by:

$$\Pi(\mathcal{C}_K, B) = \Big\{\tau \mid \tau : 2^{\mathcal{C}'} \rightarrow \{\mathsf{true}, \mathsf{false}\} \text{ where } \mathcal{C}' \subseteq \mathcal{C}_K \text{ and } |\mathcal{C}'| = B\Big\}$$

The size of this parameter space is $|\Pi(\mathcal{C}_K, B)| = \binom{K}{B} \times 2^{2^B}$. This space can be very large to explore for large values of B. In the next subsection (Subsec. 4.2.2), we discuss another assembly procedure (consensus building) used in EM-Synth that has a smaller parameter space but does not consider all Boolean combinations, as done in this assembly procedure.

4.2.2 Consensus of EM rules

This assembly procedure also builds a combination of synthesized EM rules or **GBF**s in $\mathcal{C}_\varphi = \{\varphi_1, \varphi_2, \ldots, \varphi_n\}$. But, unlike Boolean combinations, consensus building focuses on combinations of rules in a specific form i.e., it builds larger rules of the form:

$$\texttt{count\_true}\left(\varphi'_1, \varphi'_2, \ldots, \varphi'_B\right) \geqslant C$$

where (1) $\varphi'_1, \varphi'_2, \ldots, \varphi'_B$ are B EM rules or **GBF**s from $\mathcal{C}_\varphi$, (2) `count_true` represents the function that counts how many of these rules output `true` when evaluated on an example and (3) C is an integer between 1 and B , inclusive. Intuitively, this assembly procedure generates a rule that builds a consensus of at least C out of B rules when classifying an example. Note that `count_true` can be expressed as a Boolean formula in terms of the EM rules $\varphi'_1, \varphi'_2, \ldots, \varphi'_B$ as shown below:

$$\begin{aligned} &\texttt{count\_true}\left(\varphi'_1, \varphi'_2, \ldots, \varphi'_B\right) \geqslant C \\ \equiv &\bigvee_{\{i_1, i_2, \ldots, i_C\} \subseteq \{1,2,\ldots,B\}} \left(\varphi'_{i_1} \wedge \varphi'_{i_2} \wedge \ldots \wedge \varphi'_{i_C}\right) \end{aligned}$$

Hence, the assembled function $\texttt{count\_true}\left(\varphi'_1, \varphi'_2, \ldots, \varphi'_B\right) \geqslant C$ is a **GBF** since each of the $\varphi'_1, \varphi'_2, \ldots, \varphi'_B$ are also **GBF**s. This confirms that this assembly procedure is sound.

In practice, there may be many rules in $\mathcal{C}_\varphi$ i.e., n may be large. In that case, instead of looking at all possible subsets of rules in $\mathcal{C}_\varphi$ of size B, we assume that the assembly procedure is provided with a subset $\mathcal{C}_K \subseteq \mathcal{C}_\varphi$ of K rules. Given these hyper-parameters $\mathcal{C}_K, B$, the parameter space $\Pi(\mathcal{C}_K, B)$ for this assembly procedure is given by:

$$\Pi(\mathcal{C}_K, B) = \left\{(\mathcal{C}', C) \mid \mathcal{C}' \subseteq \mathcal{C}_K,\ |\mathcal{C}'| = B \text{ and } 1 \leqslant C \leqslant B\right\}$$

Note that the size of the parameter space $|\Pi(\mathcal{C}_K, B)| = \binom{K}{B} \times B$. This size is considerably

smaller than the corresponding size of the parameter space for the Boolean-combinations-based assembly procedure (Subsec. 4.2.1) because this assembly procedure considers only a fraction of Boolean combinations considered by the previous assembly procedure. But, consensus building can be run in practice with larger hyper-parameter bounds K, B, thereby giving it more opportunity to explore.

Chapter 5

Best assembly tuning

In this chapter, we discuss the techniques used in SWAPPER and EM-Synth to explore the parameter space Π for the parameter π passed to the *Assembly* procedure (discussed in Chapter 4). We start with a discussion of combinatorial auto-tuning (Sec. 5.1) of the subset selection and ordering of synthesized rules in SWAPPER using a black-box optimization tool called OpenTuner [12]. We also discuss the RS-SYNTHCOMP and RS-CONSENSUS algorithms (Sec. 5.2) used in EM-Synth to find optimal assembly of the synthesized EM rules.

5.1 Combinatorial auto-tuning in SWAPPER

In this section, we discuss the auto-tuning step of the SWAPPER system (Fig. 1-2). This step identifies the best parameter π for the assembly procedure (Chapter 4) to generate an optimal simplifier f i.e., it generates a simplifier f that minimizes an empirical performance metric based on the running time of solving simplified versions of the input problems (corpus of formulas) with the SKETCH solver. Given a set of synthesized rewrite rules $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$, the parameter space Π to be explored is given by (Sec. 4.1):

$$\Pi = \Big\{ (\mathcal{C}_f, \sigma_f) \mid \mathcal{C}_f \subseteq \mathcal{C} \ \wedge \ \big(\sigma_f \text{ is a permutation of the set } \mathcal{C}_f\big) \Big\}$$

and for n components, the size of the parameter space can be estimated to be approximately $e \times n!$ where $e \approx 2.71828$ is Euler's number. The size of the parameter space grows as quickly as the factorial function with the value of n, and it becomes really hard to enumerate all the

parameters efficiently. So, instead of using enumeration we use an off-the-shelf black-box optimization tool called OpenTuner [12] to smartly explore this space of parameters. Note that we do not guarantee global optimality of the simplifier, but instead we will generate a locally optimal simplifier that performs the best among all parameter values that we search through.

In the rest of this section, we discuss the motivation and OpenTuner configuration for auto-tuning the synthesized rules in SWAPPER.

Motivation for auto-tuning in SWAPPER

There are two motivations for auto-tuning the set of synthesized rewrite rules in SWAPPER instead of using all of them in an arbitrary order: (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 auto-tuning methods to automatically discover optimal sets and ordering of conditional rewrite rules.

Using OpenTuner for auto-tuning in SWAPPER

SWAPPER uses OpenTuner [12], a machine-learning-based off-the-shelf auto-tuner to tune the parameters of the assembly of synthesized rewrite rules in SWAPPER. As an input to OpenTuner, SWAPPER provides an optimization function **fopt**(f). The optimization function takes a simplifier f as input and emits the actual performance of the solver on formulas (from the corpus) simplified by f. This phase is comparable to algorithm configuration [13, 59], which has been used for tuning parameters for SAT solvers [60]. 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 [12] as a black box.

SWAPPER uses OpenTuner [12] to auto-tune the set of rules according to this performance metric (based on time). 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.

OpenTuner configuration

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 floating-point number. This number corresponds to performance improvement of SKETCH on the benchmarks after rewriting them using the generated simplifier (Subsec. 4.1.3). 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.

5.2 Tuning in EM-Synth

In this section, we discuss the tuning of parameters of the assembly procedure in EM-Synth. Before delving into the details, we first discuss the optimization problem that this tuning step (of the EM-Synth system shown in Fig. 1-5) solves.

5.2.1 EM-GBF optimization problem

In the **EM-GBF** rule-synthesis problem discussed in Subsec. 2.3.1, we require that the synthesized EM rule or **GBF** satisfies constraints from all provided positive and negative examples. In this section, we describe the **EM-GBF** optimization problem that corresponds to finding an EM rule to maximize an input metric μ that measures the *quality* of an EM rule. In the EM-Synth system, we want to generate a high-quality EM rule as a general Boolean formula (**GBF**). To evaluate the quality of a **GBF**, we assume that the user provides a set of examples, denoted by $\mathbf{E} = \mathbf{M} \cup \mathbf{D}$, where $\mathbf{M}$ are positive examples, i.e., pairs of records that represent the same entity, and $\mathbf{D}$ are negative examples, i.e., pairs of records that represent different entities. We also assume that the user provides a metric μ as defined below.

Optimality metric μ

Consider a **GBF** φ and positive and negative examples $\mathbf{M}$ and $\mathbf{D}$. We define a metric $\mu(\varphi, \mathbf{M}, \mathbf{D})$ returning a real number in $[0, 1]$ that quantifies the *goodness* of φ. The larger the value of μ, the better is φ.

Let $\mathbf{M}_\varphi \subset \mathbf{E}$ be the set of all examples (r', s') such that r' and s' are *matched* by φ. Some candidates for optimality metric μ are:

$$\mu_{\text{precision}} = \frac{|\mathbf{M}_\varphi \cap \mathbf{M}|}{|\mathbf{M}_\varphi \cap \mathbf{M}| + |\mathbf{M}_\varphi \cap \mathbf{D}|}$$
$$\mu_{\text{recall}} = \frac{|\mathbf{M}_\varphi \cap \mathbf{M}|}{|\mathbf{M}|}$$
$$\mu_{\text{F-measure}} = \frac{2 \cdot \mu_{\text{precision}} \cdot \mu_{\text{recall}}}{\mu_{\text{precision}} + \mu_{\text{recall}}}$$

Now, we are ready to define the **EM-GBF** optimization problem.

Problem 5 *Given two relations R and S, the aligned attributes between R and S, sets* $\mathbf{M}$ *and* $\mathbf{D}$ *of positive and negative examples, a library of similarity functions $\mathcal{F}$, and an optimality metric μ, the* **EM-GBF** *optimization problem is to discover a* **GBF** φ *that maximizes $\mu(\varphi, \mathbf{M}, \mathbf{D})$.*

5.2.2 Tuning algorithms in EM-Synth

The EM rules synthesized by the rule-synthesis step (Sec. 2.3) in EM-Synth may not perform very well with respect to the metric μ by themselves. So, we assemble (Sec. 4.2) some of these rules together to get a composite **GBF** that has a better μ value than any rule generated by the rule-synthesis step. For example, if we found 3 **GBF** rules $\varphi_1, \varphi_2, \varphi_3$ with their metric μ being $0.82, 0.77, 0.64$, respectively, then the tuning step in EM-Synth may come up with $(\varphi_1 \wedge \varphi_3) \vee \varphi_2$ as the assembly of these rules as a new rule with metric $\mu = 0.87$ that is better than each of the three **GBF**s individually. We will enumerate all possible candidate combinations in the parameter space and try to find the best combination with respect to the metric μ.

We recall the two assembly procedures described in Sec. 4.2: (1) Boolean combination, (2) consensus building. The high-level strategy for tuning in EM-Synth is to:

1. First evaluate the metric μ on all synthesized EM rules $\mathcal{C} = \{\varphi_1, \varphi_2, \ldots, \varphi_n\}$.

2. Then reduce the number of EM rules to be considered for assembly to a smaller number $K < n$ for better efficiency of tuning by taking the top K EM rules based on the metric μ i.e., it will only consider the set

$$\mathcal{C}_K = \underset{1 \leqslant i \leqslant n}{\text{argmax(K)}} \ \mu(\varphi_i, \mathbf{M}, \mathbf{D})$$

for assembly where argmax(K) returns a set of K arguments that lead to the top K metric values.

3. Then choose B ($1 \leqslant B \leqslant K$) out of those K rules and exhaustively enumerate all Boolean combinations (Subsec. 4.2.1) or consensus formulas (Subsec. 4.2.2) over the chosen B rules to find the best assembled EM rule.

The parameter spaces and their sizes for the assembly procedures based on Boolean combination (Π_{COMB}) and consensus building ($\Pi_{\mathsf{CONSENSUS}}$) are given by:

$$\Pi_{\mathsf{COMB}}(\mathcal{C}_K, B) = \Big\{\tau \mid \tau : 2^{\mathcal{C}'} \to \{\mathsf{true}, \mathsf{false}\} \text{ where } \mathcal{C}' \subseteq \mathcal{C}_K \text{ and } |\mathcal{C}'| = B\Big\}$$
$$|\Pi_{\mathsf{COMB}}| = \binom{K}{B} \times 2^{2^B}$$
$$\Pi_{\mathsf{CONSENSUS}}(\mathcal{C}_K, B) = \Big\{(\mathcal{C}', C) \mid \mathcal{C}' \subseteq \mathcal{C}_K,\ |\mathcal{C}'| = B \text{ and } 1 \leqslant C \leqslant B\Big\}$$
$$|\Pi_{\mathsf{CONSENSUS}}| = \binom{K}{B} \times B$$

For the same amount of exploration time, the tuning can process larger values of B and K for consensus building than Boolean combination. In our experiments, we use the values $K = 10, B = 3$ for Boolean combination and $K = 15, B = 5$ for consensus building.

The tuning algorithms for finding the best Boolean combination and the best consensus of EM rules in EM-Synth are called RS-SynthComp and RS-Consensus respectively. Both RS-SynthComp and RS-Consensus take as input a set of K **GBF**s ($\mathcal{C}_K$) chosen from the set of all synthesized **GBF**s $\mathcal{C} = \{\varphi_1, \varphi_2, \ldots, \varphi_n\}$. They both enumerate all parameters in their respective parameter spaces (Π_{COMB} and $\Pi_{\mathsf{CONSENSUS}}$) and for each parameter instance (1) construct a **GBF** φ' using the corresponding assembly procedure (Sec. 4.2), (2) evaluate the metric $\mu(\varphi', \mathbf{M}, \mathbf{D})$, and (3) maintain the best **GBF** with maximum value of the metric μ.

Implementation notes

Our implementation of RS-SynthComp and RS-Consensus uses bit-vectors to represent evaluations of EM rules, the SymPy library [8] for formula manipulations and the Quine-McCluskey algorithm [79] to transform a truth table τ to a Boolean formula. There is one key optimization in RS-SynthComp and one key optimization in RS-Consensus that make the computation of the metric μ efficient.

RS-SynthComp enumerates all possible B combinations of EM rules from $\mathcal{C}_K$. For a fixed set of B rules, RS-SynthComp classifies every example into 2^B buckets where each bucket corresponds to a unique set of evaluations of the B rules e.g., if $B = 3, \mathcal{C}_K = \{\varphi'_1, \varphi'_2, \varphi'_3\}$ there will be 8 buckets (shown as rows below in the table) such that each example will be assigned to a unique bucket (row in the table below) based on the evaluations of the $B = 3$ functions on that example.

φ'_1	φ'_2	φ'_3	**M**	**D**
true	true	true	100	10
true	true	false	50	20
true	false	true	40	20
true	false	false	20	200
false	true	true	60	10
false	true	false	15	100
false	false	true	20	150
false	false	false	5	500

Now, RS-SynthComp can use these buckets and the relevant counts of positive and negative examples in each bucket to evaluate the metric μ for any of the 2^{2^B} possible truth tables that will assign a value to each row in the table above. This will avoid re-computation of the number of positive/negative examples that a truth table will match correctly from scratch for each of the 2^{2^B} possible truth tables. Note that this optimization only works in cases when metric μ is computed using the number of positive/negative examples that a truth table will match correctly or incorrectly. All metrics introduced in this thesis for EM-Synth have this property e.g., precision, recall and F-measure. These metrics are also widely used in industry for entity matching.

Similarly, RS-Consensus enumerates all possible B combinations of EM rules from $\mathcal{C}_K$. For a fixed set of B rules, RS-Consensus computes a map that maps a number C ($1 \leqslant C \leqslant B$) to the number of positive and negative examples that have exactly C of the B rules evaluate to `true` on those examples. Using this map, RS-Consensus can compute the metric μ efficiently for increasing values of C from 1 to B. Again, note that this optimization only works in cases when metric μ is computed using the number of positive/negative examples matched correctly or incorrectly.

Chapter 6

Shared framework infrastructure

In the overall framework (Fig. 1-1), each of the four steps is a domain-specific procedure. There are two parts of the infrastructure that are shared among different instantiations of this framework: (1) iteratively running a general-purpose solver like SKETCH with domain-specific strategies either outside the solver as a refinement loop or inside the solver as a custom synthesizer, and (2) running different parts of each procedure (and whole experiments) in parallel while satisfying the required dependencies. Having these pieces of infrastructure enables us to quickly instantiate this framework for a new application domain.

It is important to note here that for both of these parts of the infrastructure there are existing tools that use similar ideas e.g., the Python API and DPLL(T) solvers in Z3 [41] allows a general-purpose SMT solver to be used iteratively through a Python API and specialized for a particular theory, and the Apache Spark [120] cluster computing software that can efficiently run multiple tasks in parallel on a cluster. Our implementation of these two parts uses similar ideas, but they have been specialized for our framework:

1. We built a SKETCH Python library that can express the provided SyGuS grammars, constraints and bounds in SKETCH, run SKETCH instances in parallel and parse the SKETCH output as Python data-structures to let the user (or the software using this library) decide which SyGuS problem (if any) to solve next. We also use a new DPLL(T) style feature of SKETCH and allow the user to provide a custom synthesizer (as C++ code that goes inside SKETCH) to speed up each domain-specific synthesis instance.

2. We built a custom database-backed scheduler of tasks on a computing cluster that

can be used at multiple places in the instantiations of this framework while ensuring the dependencies between tasks due to the framework are satisfied. We also provide framework-specific tools for monitoring, debugging and analyzing different tasks being scheduled and run.

We use the SKETCH Python library in the synthesis-of-components step, where multiple SKETCH problems are being solved iteratively. Both SWAPPER and EM-Synth use this library to interface with SKETCH from Python – providing their grammars and constraints to this library and analyzing the output produced by this library (a synthesized rewrite rule in SWAPPER and a synthesized EM rule in EM-Synth). In EM-Synth, we also use a custom synthesizer inside SKETCH that helps speed up the synthesis process from hours to seconds.

In the overall framework (Fig. 1-1), there are many places where parallel task scheduling on a cluster can speed up the instantiated system significantly. More specifically, some candidates for parallelization are:

1. **Data processing**: In both SWAPPER and EM-Synth, the input data consists of multiple fragments (SKETCH problems in SWAPPER and individual examples in EM-Synth), and the input data needs to be processed to build secondary representations (DAG formulas with contextual information in SWAPPER and similarity function evaluations in EM-Synth). These data processing operations tend to be embarrassingly parallelizable i.e., all of them can be run in parallel across threads, processes or machines on a cluster.

2. **Specialization information extraction**: In SWAPPER, sampling of patterns can be done in parallel e.g., by sampling from each problem DAG separately and then combining the results, or multiple instances of sampling can be run on the same set of problem DAGs on different machines and the results can be aggregated afterwards. Similarly, in EM-Synth, every RANSAC iteration can be independently run and the results can be aggregated when all of them finish running.

3. **Synthesis of components**: In SWAPPER, synthesis of rewrite rules for a pattern is independent from the synthesis of rules for another pattern. These synthesis procedures can be run in parallel across multiple processes and machines.

4. **Tuning evaluation**: when evaluating an assembled function f and computing the

numerical score of the input data, the computation can be parallelized across the fragments of the data (different input problems in SWAPPER and different examples in EM-Synth).

5. **Assembly parameter search while tuning**: when using OpenTuner or a simple enumeration, computation of the function f with different parameters can be done in parallel along with its evaluation. This is valid in both SWAPPER and EM-Synth.

6. **Running different instances of the overall framework**: To use the framework on different input data sets, we can run the full systems in parallel across different machines. This is particularly important when doing cross-validation (done in both the SWAPPER and EM-Synth systems).

Chapter 7

SWAPPER system evaluation

In this chapter, we present the high-level overview of the implementation (Sec. 7.1) of the SWAPPER system and present the aims and the results of the experiments (Sec. 7.2).

7.1 System Design & Implementation

Our implementation of SWAPPER consists of four parts corresponding to the four phases in the framework (Fig. 1-2) and three libraries that are used in different phases. We describe implementation details of the system for experiments conducted in the thesis and describe briefly how various software components may be used for other applications.

A full overview of our implementation of SWAPPER is provided in Fig. 7-1. Each software component of the system is presented in a red box (e.g., "Final evaluation"). The inputs to the system are presented in the green boxes (e.g., "Performance function"). Blue boxes represent intermediate outputs of a component of the system (e.g., "Rewrite rules"). These outputs may be fed to another component as an input. Orange boxes represent libraries (e.g., "Parallel jobs library") that are employed inside multiple software components and can be used for other standalone applications as well. We describe each component in detail below.

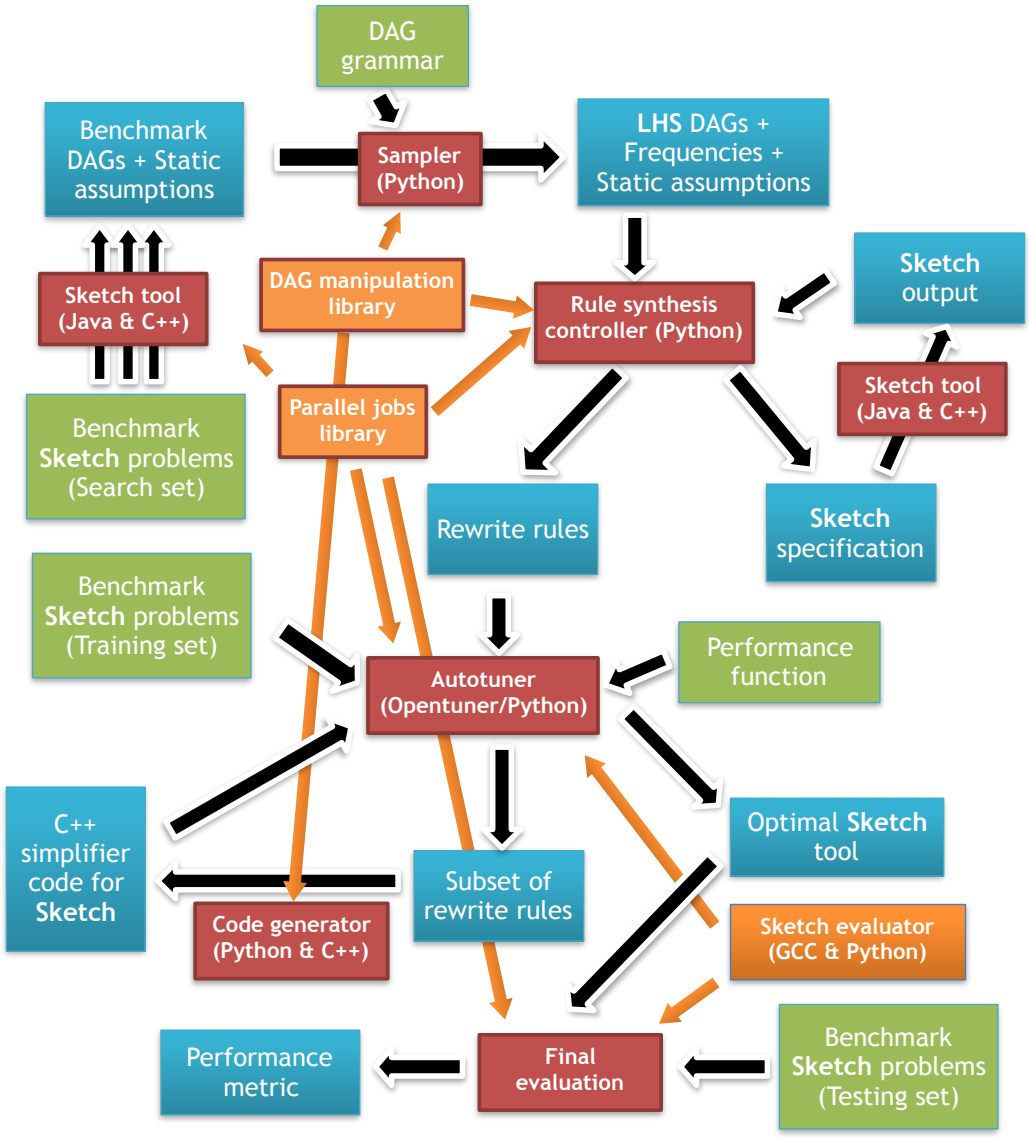

Figure 7-1: SWAPPER implementation and experiments overview

Benchmark SKETCH problems

As described further in Subsec. 7.2.1, we use SKETCH problems from two domains as benchmarks. In the experiments infrastructure, these problems are divided into three sets: Search set, Training set, and Testing set. These sets feed into: (1) generation of the benchmark formulas represented as directed acyclic graphs (DAGs) inside SKETCH with static assumptions (Sec. 2.2) (2) auto-tuning the subset and ordering of rules to be chosen for the final

simplifier (3) evaluating the final simplifier, respectively.

Sketch tool (Java & C++)

We used the existing version of the SKETCH tool without using the default simplifier to generate the benchmark DAGs. Once we generate a simplifier, we augment the same tool to obtain a potentially *optimal* SKETCH *tool*. The existing version of SKETCH is also used to solve the synthesis problems (converted to SKETCH specifications) arising from the *Rule-synthesis controller*.

DAG manipulation library

This library (written in Python) has data structures for efficient maintenance and manipulation of DAGs and conditional rewrite rules. This library enables the sampler and the rule-synthesis controllers to maintain the DAGs in a flexible data-structure and canonicalize them with hashing [80] to minimize memory usage. This library is also used in the Code generator to perform rule generalizations (Subsec. 4.1.2).

Sampler for pattern matching

Pattern matching in SWAPPER is done using the novel representative-sampling algorithm described in this thesis (Sec. 3.2). We implemented this algorithm as a generic Python tool that can be used for sampling from DAGs with any labels. This tool takes in the DAG grammar with potentially some extra labels for the nodes and maintains those labels while sampling. It outputs a ranked list of patterns based on their frequencies of occurrence in the samples along with the extra labels preserved across samples. In our case the extra labels correspond to the static assumptions obtained from the SKETCH tool.

Rule-synthesis controller

Rule synthesis in SWAPPER is done with a hybrid approach as described in Subsec. 2.2.4. We implemented this controller in Python and interfaced this with SKETCH using a custom generator (written in Python) of SKETCH specification files and a parser (again written in Python) of the SKETCH output.

Parallel jobs library

This library implemented in Python provides two interfaces for running multiple jobs in parallel on a cluster as explained in Chapter 6. This library is used in (1) generation of multiple benchmark DAGs from SKETCH problems in parallel (2) running synthesis algorithms for different patterns in parallel (3) running SKETCH instances in parallel for evaluating a simplifier in the autotuner or during the final evaluation.

Autotuner

The autotuner is a standalone Python script that uses OpenTuner [12] in the context of finding an optimal subset and permutation of the synthesized rules. OpenTuner takes as input a black-box *performance function* and the space of parameters to be optimized. In our formulation, the black-box function corresponds to the average running time of all benchmarks in the Training set, and the optimization parameters correspond to choice of a subset and ordering of the synthesized rewrite rules.

Code generator

The code generator is a Python script that takes multiple rewrite rules as input and outputs C++ code for the simplifier that can be compiled with the SKETCH tool to produce a potentially optimal version of the SKETCH tool. The output simplifier is constructed to apply the provided rules in the specified order. Moreover, the code generator performs the optimizations for rule generalization and pattern matching as described in Subsections 4.1.2 and 4.1.3.

SKETCH evaluation library

This library implemented in Python is used to compile a provided simplifier to get a modified version of the SKETCH tool and then profile this SKETCH tool on a given set of benchmarks. It stores all metrics and outputs in a shared database and a shared file system respectively. The performance function is computed in the autotuner and during the final evaluation by interfacing with this library.

7.2 Experiments

In order to test the effectiveness of SWAPPER, 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 consists of simplifications based on (a) rewrite rules that can be expressed in our framework (Sec. 2.2), (b) constant propagation, (c) structure hashing [80], 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 SWAPPER's auto-generated rewrite rules on top of the Baseline simplifier.

Now, we elaborate on the details of the experiments.

7.2.1 Domains and Benchmarks

Domain	Benchmark DAGs Used	Avg. Number of Terms
AutoGrader	45	23289
Sygus	22	68366
SAT encodings	70	6504

We investigated benchmarks from two domains of SKETCH applications. Sygus corresponds to the SyGuS competition benchmarks translated from SyGuS format to Sketch specifications [10], and AutoGrader ones are obtained from students' assignment submissions in the Introduction to Programming online edX course [100]. 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 are 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 the auto-tuning phase. Using these cutoffs, the total number of usable benchmarks for AutoGrader domain was reduced from 2404 to 45 and for Sygus from 309 to 22. We also performed a case study on SAT-encodings benchmarks [62] translated from synthesis specifications for SMT to SAT encoding rules.

7.2.2 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 each. A worst-case estimate of the cost of computation done by SWAPPER based on our experiments using the Amazon Web Services [1] 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 $\times$ 150	20	$23
Sygus	2 hours	1 hour $\times$ 5	0.1 $\times$ 150	22	$26

In essence, SWAPPER can be used to automatically synthesize a simplifier for a very reasonable cost (less than $50) spent on computation (around what one would pay a good developer for an hour's worth of work). Note that these wait times (around 20-22 hours or a day to get a simplifier) can be improved significantly by: (a) parallelization of the

pattern-finding phase (b) setting a timeout for evaluation runs that are guaranteed to be worse than existing good runs and (c) increasing the parallelism available to SWAPPER.

7.2.3 SWAPPER Performance

To test the performance of SWAPPER on SKETCH benchmarks from a particular domain, we divided the corpus into three disjoint sets randomly (SEARCH, TRAIN, 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, the 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 the TRAIN set. We achieved this by exchanging the 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 obtains 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. We obtained 301 rules for the AutoGrader domain and 105 rules for the 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

The 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 the Hand-crafted simplifier (Figure 7-2). On DAGs obtained after using the Auto-generated simplifier, on average, the SKETCH solver performed better than on those obtained by using the Hand-crafted simplifier (Figure 7-3): (1) the 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 the Auto-generated simplifier were 13s (AutoGrader) and 8s (Sygus) as compared to 20s and 21s respectively

for the Hand-crafted simplifier. Figures 7-2 and 7-3 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

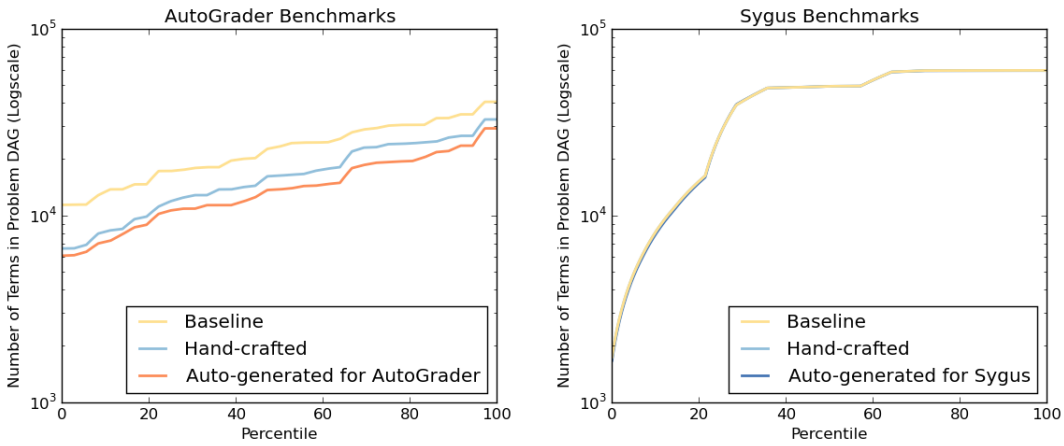

Figure 7-2: Change in sizes with different simplifiers

Auto-generated simplifier reduced the size of the problem DAGs by 38.6% (AutoGrader) and 1.6% (Sygus) on average (Figure 7-2). Application of Auto-generated simplifier results in huge improvements in running times for the 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 7-3).

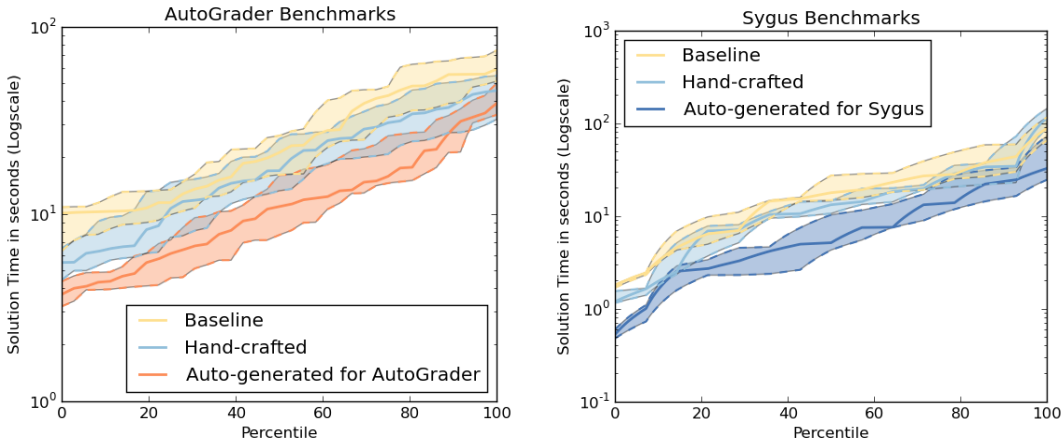

Figure 7-3: Median running-time percentiles with quartile confidence intervals

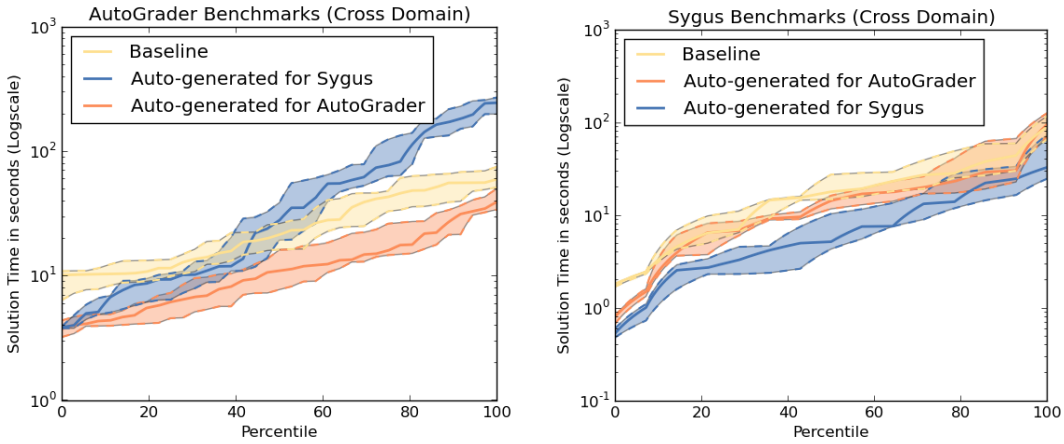

Figure 7-4: Domain specificity of the Auto-generated simplifiers: Time distribution

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 the Auto-generated simplifier obtained from Sygus increased the SKETCH running times drastically on a few AutoGrader benchmarks when compared to the application of the Baseline simplifier (Figure 7-4) and resulted in SKETCH running slower than after application of the Auto-generated simplifier obtained from the AutoGrader domain. Application of the Auto-generated simplifier generated for AutoGrader domain reduced the running times of SKETCH solver on average as compared to the Baseline on the Sygus benchmarks, but the times were still far away from the performance gains obtained by application of the Auto-generated simplifier generated for the Sygus domain (Figure 7-4). This validates

our hypothesis of these generated simplifiers being very domain-specific.

7.2.4 SAT Encodings Domain

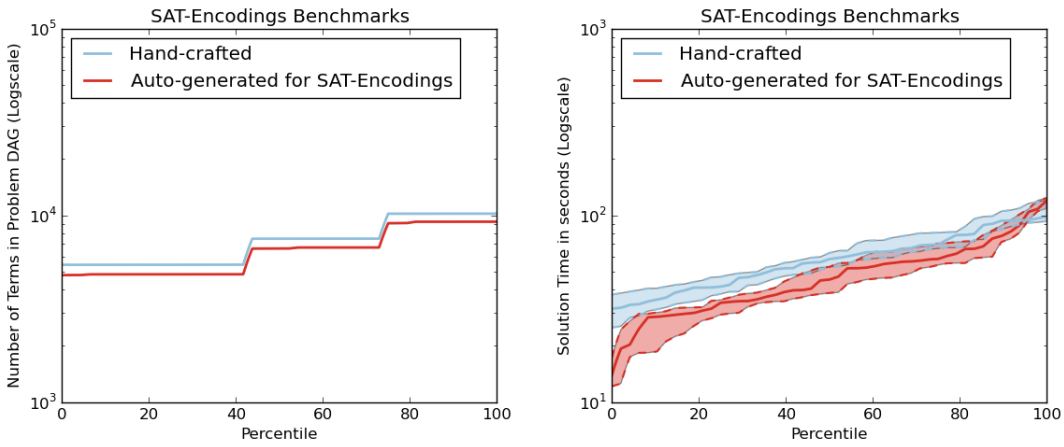

Figure 7-5: SAT-Encodings domain case study

We performed an additional case study using SWAPPER on problems generated during synthesizing optimal CNF (SAT) encodings [62]. 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. 7-5. SWAPPER generated 117 rules and, on average, the SKETCH solving time reduced from 58.8s to 51.1s, and the DAG sizes were reduced by 11%.

7.2.5 Analysis of Generated Rules and their Impact

In Fig. 7-2 the average size reduction for Sygus benchmarks is 1.1%, which isn't clearly visible in the graphs, but still these rewrites enable the Sketch solver to run faster. Sygus benchmarks are written at a level of abstraction that is very close to the DAGs in Sketch. By contrast, the AutoGrader benchmarks come from Python code that is automatically translated to Sketch code which is then automatically translated to formulas. Each step in this translation introduces inefficiency, so there is a lot more opportunity for optimization.

We show two example rules generated for the same LHS pattern in Fig. 7-6. The rule has both integers and arrays of integers as inputs. The supplementary C-like code is only provided for better understanding of the semantics of the pattern – the actual semantics when

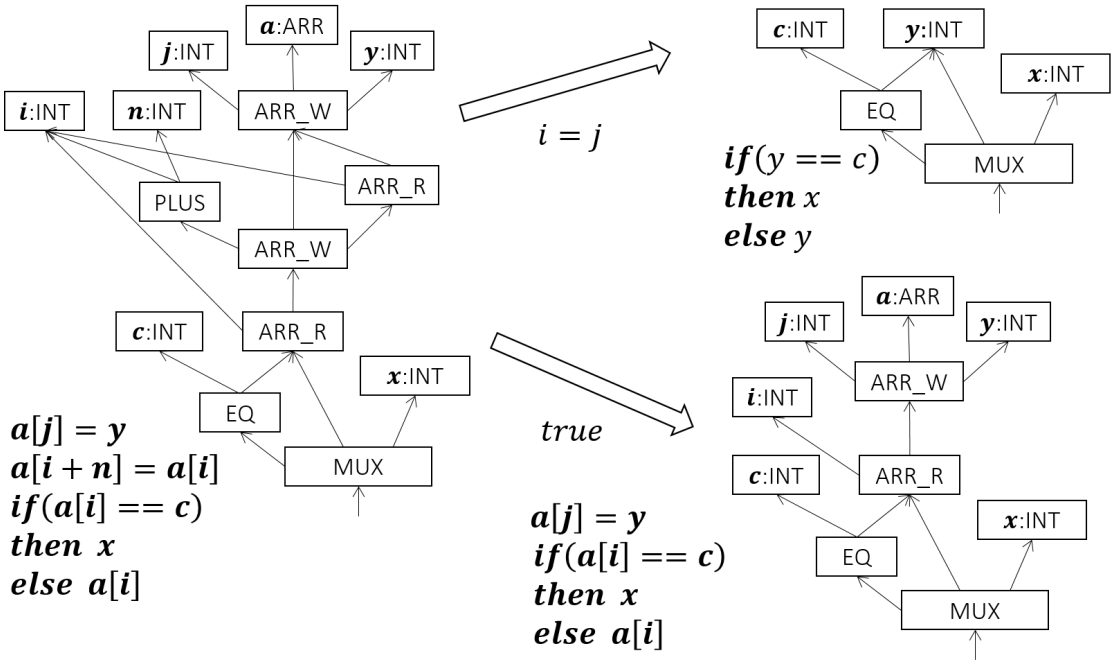

Figure 7-6: An example rule generated for AutoGrader benchmarks

interpreting the pattern internally in SKETCH are different from the C-like interpretation and follows the interpretation based on the SMT theory of arrays. We can see from the example that: (1) it is a large LHS pattern that is hard for a human to recognize as a potential source of an optimization, (2) the generated rewrite rules are complicated to reason about, and (3) there are two rules with different strengths of the predicates – this showcases the trade-off between applicability (strength of the predicate) and the extent of the simplification (size reduction from LHS to RHS) because of the rewrite rules.

Chapter 8

EM-Synth system evaluation

In this chapter, we present some additional details of the RuleSynth algorithm along with a high-level overview of our implementation of the EM-Synth framework. We also discuss the implementation of the platform we built in order to conduct experiments and compare different EM techniques with RuleSynth. We discuss the setup and results of these experiments as well.

8.1 Algorithms and optimizations in EM-Synth

In this section, we first describe some algorithmic optimizations employed in EM-Synth and the motivations behind using them. Then, we setup the terminology for the algorithms used in the experiments presented later in this chapter.

8.1.1 Incremental grammar bounds in RS-CEGIS

To make sure that the generated rules are small and concise, we modify the RS-CEGIS algorithm (Subsec. 3.4.1) to iteratively adjust the grammar bound on the number of attribute-matching rules (N_a) as it runs, starting with rules of size 1 and growing up to N_a, so that it prefers smaller rules when they can be found. To be more precise, we introduce the following loop in Algorithm 5 replacing line 6.

RS-CEGIS uses an optimized version of this loop where in CEGIS iteration $i \geqslant 1$, the initial value of n_a is set to the value of n_a used to synthesize φ_{i-1} in the previous CEGIS iteration (instead of starting with $n_a = 1$). Since the set of examples being considered in iteration i is a superset of examples considered in iteration $i-1$, if for any n_a **Synth** could

Procedure Incremental grammar bounds

```
1 n_a ← 1                                          // attribute-matching rules bound n_a
2 while n_a ⩽ N_a do
3     φ_i ← Synth(G_GBF(n_a, N_d), E_SYN, F)
4     if φ_i = null then                                  // Unsatisfiable Synth
5         n_a ← n_a + 1                                        // try larger n_a
6     else
7         break
```

not find a **GBF** in iteration $i-1$, then for the same n_a it will not be able to find a **GBF** that matches all the examples in iteration i.

8.1.2 Sampling: bias in picking examples in RS-CEGIS

In CEGIS iteration i, the RS-CEGIS algorithm (Subsec. 3.4.1) tries to primarily choose an example that is currently not being matched correctly. This guides the resulting **GBF** towards higher accuracy on the example set by making more and more examples match correctly. For optimality metrics like $\mu_{\text{F-measure}}, \mu_{\text{precision}}, \mu_{\text{recall}}$ it is important to focus on finding **GBF**s that maximize the number of positive examples being matched correctly. Note that if the set of examples is largely only negative examples (which is the case in our benchmark datasets) then the likelihood of most of the chosen examples being negative is high. This may result in the algorithm missing certain positive examples for smaller CEGIS cutoffs (K_{CEGIS}) and thereby finding a solution with possibly lower μ even when the accuracy is high. Hence, in RS-CEGIS we eliminate this bias based on the actual distribution of positive and negative examples and replace it with a 50-50 chance of choosing a positive or negative example, i.e., the **sample** routine (lines 3 and 14 in Algorithm 5) is modified as described above.

8.1.3 Algorithms for entity matching using EM-Synth

In our evaluation of EM-Synth, we compare three different algorithms built using the EM-Synth system. Two of them, (1) RS-SynthComp and (2) RS-Consensus, use different assembly and tuning methods and have already been discussed in Sec. 5.2. The third algorithm RuleSynth is a simple extension of the RS-RANSAC algorithm described in Subsec. 3.4.2 where RuleSynth simply chooses the best performing EM rules synthesized

across all CEGIS and RANSAC iterations without combining multiple EM rules together. Note that RuleSynth can also be seen as an instantiation of our overall framework (Fig. 1-1) where the assembly of components $\mathcal{C}$ corresponds to

$$Assemble(\pi, \mathcal{C}) = \operatorname*{argmax}_{\varphi \in \mathcal{C}} \mu(\varphi, \mathbf{M}, \mathbf{D})$$

and the output of the assembly is independent of the parameter π, so there is no need for any tuning of parameters.

In our experiments, we compare RuleSynth, RS-SynthComp and RS-Consensus with other entity-matching techniques.

8.1.4 Bucketing-based optimized EM-rule testing

The final EM rule φ obtained from EM-Synth (using one of the RuleSynth, RS-SynthComp, and RS-Consensus algorithms) will be applied to a pair of records (r, s) for $r \in R, s \in S$ to figure out whether or not these records match, where $R[\mathsf{A_1}, \mathsf{A_2}, \ldots, \mathsf{A_n}]$ and $S[\mathsf{A'_1}, \mathsf{A'_2}, \ldots, \mathsf{A'_n}]$ are two relations with corresponding sets of n aligned attributes $\mathsf{A_i}$ and $\mathsf{A'_i}$ for $i \in [1, n]$ (same notation as in Subsec. 2.3.1).

In practice, the rule may be applied to large tables. Suppose that there are N_R, N_S records in relations R and S respectively. To evaluate the EM rule as a *query* on all pairs of records, a naïve approach would enumerate all such pairs and individually apply the rule φ $N_R \times N_S$ times. A faster approach would be to use "bucketing" of records [45] (based on a locality-sensitive hashing function [118]) from both relations so that φ will only have to be evaluated on a record (r, s) when both r and s belong to the same bucket. This can potentially reduce the number of records to compare from $N_R \times N_S$ to $O(min(N_R, N_S))$.

Note that this approach can only be used when a **GBF** φ has a desired structure. To be more specific, we require:

1. The **GBF** φ can be decomposed as:

$$\varphi \equiv \left(r[\mathsf{A_i}] \approx_{(f,\theta)} s[\mathsf{A'_i}]\right) \wedge \varphi'$$

   i.e., we are able to bucket those records together that may satisfy the first part $(r[\mathsf{A_i}] \approx_{(f,\theta)} s[\mathsf{A'_i}])$ of the rule. Any records that will be mismatched by the first part

will be put in different buckets.

2. The similarity function f has to be one of the few functions that have corresponding locality-sensitive-hashing (LSH) function families allowing indexing based on a threshold and hence, allowing bucketing e.g., Equality, Jaccard and Cosine similarity functions. Examples of more such similarity functions are presented in [118]. At least 7 out of 30 functions considered in our experiments have such a corresponding LSH family.

In Fig. 8-1, we present some examples of similarity functions and their corresponding hashing-function families from the literature.

Similarity function	**Hashing function family**
Equality	Identity
Jaccard Similarity	Min-Hash [25, 26]
	Min-Max Hash [64] & others [118]
Cosine Similarity	Concomitant LSH [46]
	Cross-polytope LSH [11]

Figure 8-1: Some examples of similarity functions with their corresponding hashing-function families

In practice, this approach can reduce the time taken to run the query as shown by the experiments in this thesis (*Exp-7* in Sec. 8.4). It is important to note here that the hashing-function families discussed here are probabilistic in nature and the bucketing may result in some positive example pairs being mapped to different buckets (making them false negatives) – but such an occurrence is very unlikely and can be mitigated by increasing the amount of randomness used in the computation of these LSH families. In our experiments, we did not observe such false negatives when bucketing with a hashing-function family.

8.2 System design and implementation

We implemented EM-Synth with both of the optimizations mentioned above and built a platform around it to compare different EM techniques. The implementation of this overall platform consists of three parts: feature processing, EM algorithms and experiment infrastructure. Fig. 8-2 shows a high-level diagram of the EM-Synth system. The dark red boxes

(e.g., "Pruning negative examples") represent procedures or steps in the EM-Synth system, the green boxes (e.g., "Metric μ") represent the inputs to the system, the blue boxes (e.g., "Feature vectors") represent intermediate objects that can be fed as inputs to other procedures, and the orange boxes (e.g., "Training module") are the generic modules that can be replaced with or instantiated with different parts based on what EM technique is being used. We explain these in detail below.

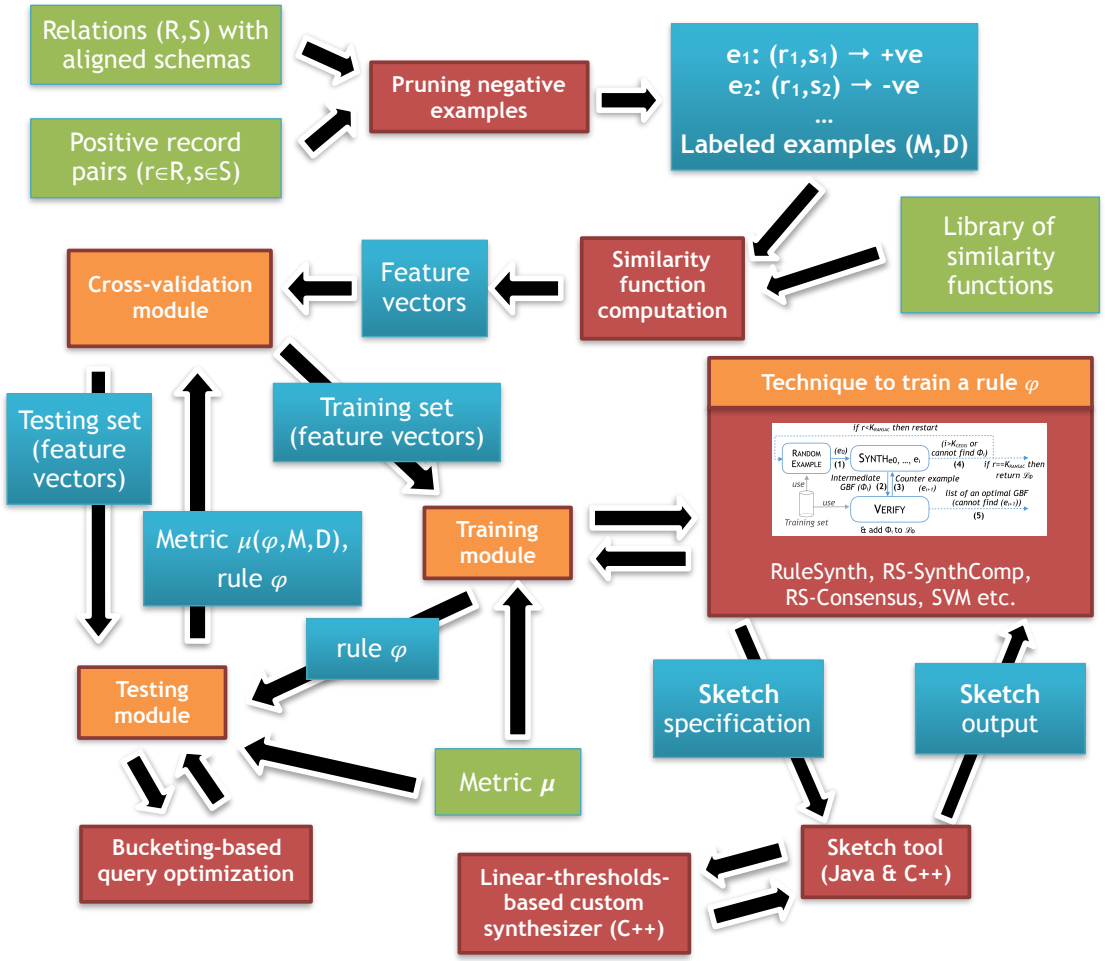

Figure 8-2: EM-Synth implementation and experiments overview

8.2.1 Feature processing

This part of the infrastructure was used to create, maintain and manage feature vectors shared by all EM techniques. The feature vectors in the context of EM-Synth are a mapping from examples to a list of evaluations of all similarity functions on all aligned attributes i.e.,

given two relations $R[A_1, A_2, \ldots, A_n]$ and $S[A'_1, A'_2, \ldots, A'_n]$ with aligned attributes (A_i is aligned with A'_i), a library of similarity functions $\mathcal{F}$ and positive ($\mathbf{M}$) and negative ($\mathbf{D}$) examples $\mathbf{E} = \mathbf{M} \cup \mathbf{D}$, the feature-vector mapping for $(r, s) \in E$ is given by

$$fv(r, s) = [f(r[A_i], s[A'_i]) \text{ for } i \in [1, n], f \in \mathcal{F}]$$

Pruning negative examples

For all datasets except the Cora dataset, we had access to only positive examples $\mathbf{M}$ and we had to construct the set of negative examples $\mathbf{D}$ from the relations R, S and $\mathbf{M}$. To ensure that negative examples in $\mathbf{D}$ are quite different from each other, we took the Cartesian product of the relations and pruned pairs with high Jaccard tri-gram similarity values [68,69]. This pruning is similar to the method adopted in SIFI [116]. We varied the similarity threshold across datasets to control the number of negative examples.

Similarity-function computation

We use a set of 29 similarity functions that were also used in the SIFI project [117]. This set includes functions from the Simmetrics library (`https://github.com/Simmetrics/simmetrics`) written in Java and functions implemented by authors of SIFI in C++. We also treat `Equal` and `noNulls` as two similarity functions that evaluate to 0 or 1. We implemented a Python wrapper around the Java and C++ code using the Pyjnius [4] and Boost Python [2] libraries. We use the outputs of these similarity functions rounded to a finite precision of 3 decimals.

This computation was done in parallel, and the result was stored in a database using the custom cluster computing software built for our framework (Chapter 6).

Example sampling: for cross-validation and limited-data experiments

We use a random sampler written in Python that maintains the fraction of positive examples when sampling $X\%$ of the data from training data i.e., the fraction of positive examples in the sample should roughly be the same as the fraction of positive examples in the full training data. We also maintain these samples across various techniques so that we can do a fair comparison across different EM techniques.

8.2.2 EM algorithms

We built a generic platform that can test various EM algorithms on our datasets. The interfaces provided by the platform are abstracted as three libraries or modules called the *cross-validation module*, the *training module* and the *testing module*. These modules implement a generic cross-validation framework where the cross-validation module either samples the data or uses the existing samples of the training data to perform k-fold cross validation. It provides the training set to the training module, which uses an EM technique (separate box) to generate a rule or a classifier φ. The testing module then evaluates φ and records the metric in the cross-validation module. This process continues k times. The statistics corresponding to various runs e.g., time taken, the metric μ values, size of the classifier and other metrics are stored in a database for easy access.

Implementation of EM-Synth based algorithms

The three algorithms (Subsec. 8.1.3) based on EM-Synth are implemented on top of the RS-RANSAC algorithm (Subsec. 3.4.2) as separate Python scripts. The RS-RANSAC and RS-CEGIS algorithms are instantiations of the basic training module that uses feature vectors as a proxy for the examples and selects the appropriate feature vectors as per its choice of the examples. RS-CEGIS also uses the testing module to find the counter-examples (i.e., the feature vectors corresponding to those that mismatch with the previously synthesized **GBF**). RS-RANSAC is a simple loop in Python outside RS-CEGIS.

Synth routine: Python to Sketch bridge

RS-CEGIS uses the Sketch solver (implemented in Java and C++) along with the custom synthesizer implemented inside Sketch in C++ to synthesize the EM rules. EM rules are implemented as a nested list data-structure in Python. We build a bridge between the Sketch tool and the Python script interpreting and testing the synthesized EM rules with the help of a Sketch specification generator class in Python and a Sketch output parser in Python. These two components enable quickly running the Sketch solver and parsing its output. The custom synthesizer is also provided a table (as a text file) of similarity-function evaluations only for the relevant examples used in the Sketch specification.

8.2.3 Experiment infrastructure

All experiments were run on multiple machines with Ubuntu 14.04 OS, 32 GB RAM and 16-core 2.3 GHz CPUs. We discuss the algorithms implemented outside the EM-Synth system below.

Other algorithms in the platform

We built infrastructure for comparing the EM-Synth algorithms with other techniques that could be integrated into our platform easily. In particular we implemented the following techniques:

- **SVM, Gradient tree boosting**: we use the Python scikit-learn library for these [85].
- **Random forests, Decision trees**: we use the Weka library [109] for these techniques. These implementations were obtained from the authors of [54].
- **SIFI**: we use the C++ implementation obtained from the authors of [117].

We transform the feature vectors to the required formats of the two libraries before running the experiments with these techniques. We also use a simple grid search [9] to tune the hyper-parameters of these ML algorithms e.g., we try different kernels and penalty hyper-parameters for SVM [5] and choose the values that perform the best in terms of the metric μ.

LSH-based bucketing

We use a MinHash locality-sensitive hashing (LSH) index [3,74] to bucket potentially similar strings of a given attribute with respect to Jaccard similarity of the sets of their bi-grams and a given threshold. We use the datasketch Python library for implementing this MinHash LSH scheme [3] and produce all the possible pairs of examples inside each bucket to be checked for a match using a rule in the next step.

Deployment of multiple experiments on a cluster

Similar to Swapper, we use the parallel jobs library (Chapter 6) built by us to run various experiments in parallel. Moreover, for the same dataset, multiple instances of training are run on the same machine for different folds. The result of each experiment run is stored in

a central database, and the full log of the experiment is stored in a shared file system. This allows EM-Synth to quickly run multiple experiments and different parts of each experiment in parallel.

8.3 Experimental setup

We discuss the experimental setup before we present the results in the next section. The key questions we answer with our evaluation are:

1. How do our rules compare in interpretability and accuracy to other interpretable models? (Exp-1, Exp-2);
2. How do they compare in accuracy to expert-provided rules? (Exp-3);
3. How do they compare in accuracy to non-interpretable models, such as SVMs? (Exp-4);
4. How do we perform when using limited training data? (Exp-5);
5. Can RuleSynth discover rules in reasonable amounts of time? (Exp-6);
6. How efficient are the RuleSynth rules compared to non-interpretable ML models when applied to large datasets? (Exp-7); and
7. How important is the new special-purpose synthesizer (Subsec. 2.3.3) inside Sketch for solving the **EM-GBF** rule-synthesis problem (Subsec. 2.3.3)? (Exp-8).

We describe the datasets, baseline approaches and the experimental results in the following sub-sections.

8.3.1 Datasets

Fig. 8-3 shows the four real-world datasets used in our evaluation. The Cora dataset has one relation, while the others have two relations with aligned schemas. Positive examples for every dataset are also given and we derive the negative examples using a Jaccard similarity based pruning as explained in Subsec. 8.2.1. Figure Fig. 8-3 also shows the average number of record pairs with at least one null value. These numbers show the importance of using the custom noNulls function in a formula because noNulls in the `if` condition enables the

synthesizer to find smaller rules for the noNulls (`then`) vs nulls (`else`) cases. Some datasets have a skewed distribution of nulls across attributes, e.g., for **DBLP-Scholar**, the attribute year has around $40K$ nulls, whereas title and authors have 0.

	#Matching Pairs	#Record Pairs	#Attr	Avg #nulls per Attr
D_C	14,280	184,659	9	92,955(50%)
D_{AG}	1,300	97,007	4	22,583(23%)
D_{LF}	6,048	341,244	10	99,629(29%)
D_{DS}	5,347	112,839	4	12,685(11%)
D_C =**Cora**, D_{AG} =**Amazon-GoogleProducts** D_{DS} =**DBLP-Scholar**, D_{LF} =**Locu-FourSquare**				

Figure 8-3: Dataset statistics

All of these datasets used in this thesis can be downloaded from this Google Drive folder[1] and the instructions for understanding the data can be found at this Google document[2]. We also briefly describe each dataset and its origin below:

1. **Amazon-GoogleProducts**: This is an e-commerce dataset that contains attributes (name, description, manufacturer, price) of certain products as presented on Amazon.com and the product search service of Google accessible through the Google Base Data API. The entities are matched across data obtained from Amazon.com and Google Base Data API. This dataset was obtained from the authors of [69].

2. **DBLP-Scholar**: This is a bibliographic dataset that contains attributes (title, authors, venue, year) about research publications collected from the DBLP server and the search engine Google Scholar. the entities are matched across data obtained from DBLP and Google Scholar. This dataset was also obtained from the authors of [69].

3. **Cora**: This is also a bibliographic dataset that contains attributes (author, title, venue, address, publisher, editor, date, volume, pages) of citations of research publications collected from multiple sources as a part of the *Cora search engine* [78]. The entities are matched across multiple citations i.e., entity matching in this dataset corresponds to deduplication of citations. This dataset was curated by and obtained from the authors of SIFI [116].

[1]The folder is available at `https://goo.gl/rCVZ79`
[2]The instructions document is available at `https://goo.gl/cGmkNJ`

4. **Locu-Foursquare**: This is a local-business-discovery dataset that contains attributes (website, name, locality, country, region, longitude, phone, postal_code, latitude, street_address) of local businesses from two online aggregators (Locu and FourSquare). We obtained this dataset and the permission to use it from Locu and curated it ourselves. This dataset was also used in the Advanced Topics in Computer Systems course at MIT[3].

8.3.2 Performance and interpretability metrics used

In our experiments, we use F-measure (Subsec. 5.2.1) as the metric to be optimized. To compare interpretability of rules produced by different approaches, we count the number of attribute-matching rules or *atoms* (Subsec. 2.3.1) in the Boolean formula representing the rule.

8.3.3 Similarity functions used

We use a set of 29 similarity functions that were also used in the SIFI project. On top of these, we also use the `noNulls` function. Note that a difference between our evaluations as compared to SIFI is in the way we handle null values – our similarity functions return 0 as long as one of the input values is null in our framework whereas for SIFI, if both values are null, the similarity functions will return 1. Figure Fig. 8-4 reports a complete list of the functions used in this evaluation.

8.3.4 Input features for ML techniques

For every example record pair, we evaluate all available similarity functions on strings from aligned attributes and construct a vector of these numerical values between 0 and 1. These vectors are used as input feature vectors for all ML techniques. For SVM, we also normalize the feature vectors to have zero mean and unit variance during training and use the same scaling while testing [7].

8.3.5 Comparisons with state-of-the-Art ML approaches

We compare the three algorithms from EM-Synth (RuleSynth, RS-SynthComp, and RS-Consensus) with decision trees, SVM [117], gradient tree boosting [34] and random

[3] `https://github.com/mitdbg/asciiclass/blob/master/labs/lab4/README.md`

1. Equal 2. noNulls 3. EditDistance
4. CosineToken 5. DiceToken 6. OverlapToken
7. JaccardToken 8. CosineGram$_2$ 9. DiceGram$_2$
10. OverlapGram$_2$ 11. JaccardGram$_2$ 12. CosineGram$_3$
13. DiceGram$_3$ 14. OverlapGram$_3$ 15. JaccardGram$_3$
16. BlockDistance 17. ChapmanLengthDeviation
18. ChapmanMatchingSoundex 19. ChapmanMeanLength
20. ChapmanOrderedNameCompoundSimilarity
21. EuclideanDistance 22. Jaro 23. JaroWinkler
24. MatchingCoefficient 25. MongeElkan
26. NeedlemanWunch 27. SmithWaterman
28. SmithWatermanGotoh 29. Soundex
30. SmithWatermanGotohWindowedAffine

Figure 8-4: Input-similarity functions ($\mathcal{F}$)

forests [54]. All ML methods solve entity matching by treating it as a binary-classification problem.

While the output from SVM lacks logical interpretability, a decision tree can be interpreted as a Boolean formula with multiple **DNF** clauses arising from traversal of paths that lead to positive classification. However, the outputs of random forests and gradient tree boosting are tedious to interpret because: (1) the output has tens to hundreds of trees that are aggregated to make the final decision, (2) each decision tree in random forests can have a large depth, resulting in thousands of nodes, making them hard to interpret individually, and (3) there are hundreds of weights associated with trees or leaves of the trees that cannot be translated to Boolean logic.

8.3.6 Comparisons with rule-based learning approaches

We compared RuleSynth, RS-SynthComp, and RS-Consensus (from Subsec. 8.1.3) against a heuristic-based approach, SIFI [117], which searches for optimal similarity functions and thresholds for the attribute comparison given a DNF grammar provided by a human expert. In contrast, the **GBF**s are automatically discovered by RuleSynth without any expert-provided structure of the rules.

8.3.7 Techniques and parameters

For all ML techniques, we used a simple grid search [9] of values for different parameters. We list the parameters being searched for below:

1. For decision trees: depth of the tree, minimum number of examples needed for a split.
2. For SVM: choice of kernel (LinearSVC or RBF) [5], the penalty hyper-parameter C in the loss function and γ hyper-parameter for RBF kernel.
3. For gradient tree boosting: the learning rate, maximum depth of a tree, maximum number of trees.
4. For random forests: maximum depth of a tree, number of trees.

For decision trees, we separately present results for depths $3, 4$ and 10 (the default configuration in Weka). For SVM, we separate the results for the two kernels. For gradient tree boosting and random forests, we present results with *small* #-atoms, i.e., 2-4 trees of depth 2-4 (so that #-atoms is bounded by 60), and *large* #-atoms (searching around the defaults in the Scikit learn [85] libraries on the grid), i.e., 5-15 depth or unlimited-depth trees for random forests and 25-100 trees with depth 2-4 for gradient tree boosting. Note that, even though these two techniques have interpretable trees, each tree or leaf has a numerical weight assigned to it that makes them hard to interpret. For SVM we use balanced class-weights as a low-effort configuration for optimizing F-measure [82]. We also ran SIFI with default configurations and grammars given by human experts as input.

We use the three algorithms from EM-Synth i.e., RuleSynth, RS-SynthComp, and RS-Consensus (from Subsec. 8.1.3). For these three algorithms, we have the following parameters with their respective default values:

1. The depth of the grammar $N_d = 4$, which is enough to represent formulas with at most 15 atoms.
2. A high $K_{\text{CEGIS}} = 1000$ (Subsec. 3.4.1) with a timeout of 15 minutes per CEGIS iteration so that the CEGIS loop runs until it finds a set of examples for which Sketch cannot synthesize a valid rule or it times out and collects all the rules obtained till then.

3. The bound $K_{\text{RANSAC}} = 5$ (Subsec. 3.4.2) to restart CEGIS 5 times and explore different underlying sets of examples.

4. The number of attribute-matching rules N_a: for RuleSynth, we set $N_a = 8$ so that it is comparable with the number of atoms in a decision tree of depth 3, and since we are not composing any of the synthesized rules together, these rules would be large enough by themselves. For RS-SynthComp we use $N_a = 5$ and combine $B = 3$ rules out of $K = 10$ rules (Subsec. 4.2.1) to generate a composite **GBF** so that in total #-atoms is bounded by 15 and is comparable with #-atoms in a decision tree of depth 4. For RS-Consensus, we use $N_a = 8$ and combine $B = 5$ rules out of $K = 15$ rules (Subsec. 4.2.2) to have similar #-atoms as the *small* gradient tree boosting and random forests (with $2 - 4$ trees of depth $2 - 4$).

8.3.8 Performance evaluation

We performed K-fold cross-validation (for K=5) on each of the datasets used, where we divided the data into K equal fractions (folds) randomly and performed K experiments. In each experiment one of the K folds was the test set while the remaining $K - 1$ folds were training. We report the average F-measure obtained across all folds on the test sets as the performance metric (Figure 8-7). Note that we use the same folds for each technique we compare, and for each fold we may find different optimal values for the parameters of the ML techniques.

For limited-training-data experiments (*Exp-5*), we randomly sample a fraction $\frac{1}{K}$ of examples (for multiple values of $K = 100, 40, 20, 10, 7, 5$) and use it for training. Each fraction $\frac{1}{K}$ corresponds to a different percentage $P\%$ of examples (i.e., $P = 1, 2.5, 5, 10, 14.3, 20$). We use the rest $\big((100 - P)\%\big)$ of the examples for testing, and we train and test on 100 such randomly selected sets for each percentage P. We report the average test-set F-measure and size of matching rules obtained across all 100 runs (with 99% confidence intervals).

8.4 Experimental results

Now that we have discussed the experimental setup, we are ready to present the experimental results.

8.4.1 Exp-1: Interpretability

By interpretability, we mean how readable and understandable the discovered rules are. We measure it as being inversely proportional to the number of *attribute-matching rules* (or atoms) present in the rule. In other words, interpretability is defined as the number of atomic similarity function comparisons with a threshold $\approx(i, f, \theta)$ in the formula representing the rule. For clarity, we represent atoms or attribute-matching rules as $\big(fn[attr] \geqslant \theta\big)$, where fn is the name of the applied similarity function, $attr$ is the name of the matched attribute, and θ is the corresponding threshold, e.g., $\mathsf{EditDistance[title]} \geqslant 0.73$ is a valid atom. The rationale behind this interpretability definition is that fewer atoms make a rule easier to read. This supports the idea that a complex **DNF** is less interpretable than a semantically equivalent but concise **GBF**. We also conduct an informal user study with 27 participants to show the correlation between the number of atoms and the user preference for more interpretable rules. In short, statistical ML methods with weights and function parameters (e.g., SVM, random forests, gradient tree boosting) and logical structures with hundreds or thousands of atoms (e.g., decision tree with depth 10) are not human interpretable. Methods with clear logical structures, such as **GBF**s, **DNF**s, and decision trees with depths 3 and 4, are human-interpretable.

Below, we present two **GBF**s, φ_{synth} and φ_{tree}, obtained by using RuleSynth and decision trees of depth 3, respectively, on record pairs from the Cora dataset. We obtained both **GBF**s on the same training set as the best rules. These rules result in average F-measures of 0.83 (φ_{synth}) and 0.77 (φ_{tree}) on test data. The **GBF** φ_{synth} demonstrates the conciseness of formulas generated by RuleSynth as compared to φ_{tree}, as φ_{synth} has only 6 atoms whereas φ_{tree} has 12 atoms. Also note that the RuleSynth rules include if/then/else clauses that allow them to be more compact that the DNF-based rules the decision tree produces.

φ_{synth} : ($\mathsf{ChapmanMatchingSoundex[author]} \geqslant 0.937$

$\bigwedge$ **if** $\mathsf{noNulls[date]} \geqslant 1$

then $\mathsf{CosineGram_2[date]} \geqslant 0.681$

else $\mathsf{NeedlemanWunch[title]} \geqslant 0.733$) $\bigvee$

($\mathsf{EditDistance[title]} \geqslant 0.73$

$\bigwedge \mathsf{OverlapToken[venue]} \geqslant 0.268$)

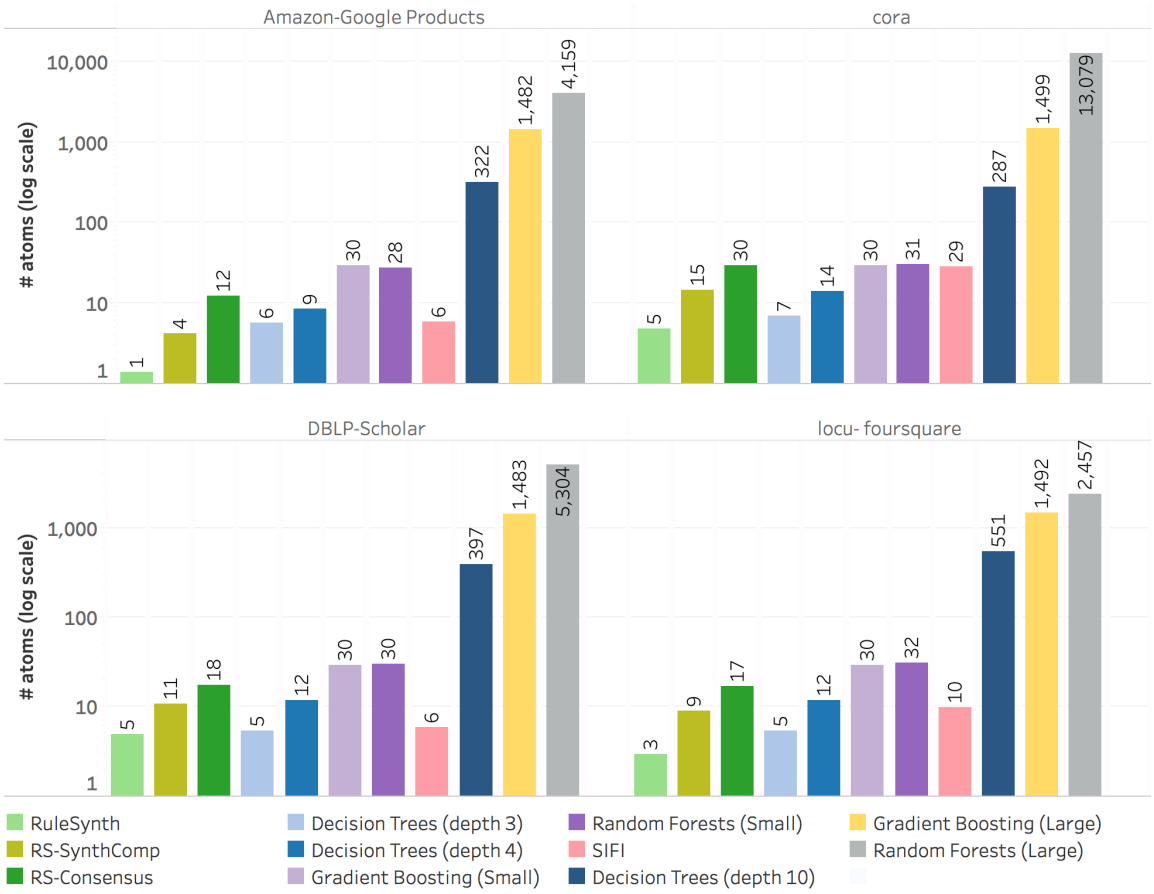

Figure 8-5: Interpretability results for 5-folds experiment (80% training and 20% testing data)

$$
\begin{aligned}
\varphi_{tree} : \big(\; & \texttt{OverlapGram}_3[\text{title}] \geqslant 0.484 \\
& \bigwedge \texttt{MongeElkan}[\text{volume}] \geqslant 0.429 \\
& \bigwedge \texttt{Soundex}[\text{title}] \geqslant 0.939 \big) \bigvee \\
\big(\; & \texttt{OverlapGram}_2[\text{pages}] \geqslant 0.626 \\
& \bigwedge \texttt{MongeElkan}[\text{volume}] \geqslant 0.429 \\
& \bigwedge \neg \big(\texttt{Soundex}[\text{title}] \geqslant 0.939 \big) \big) \bigvee \\
\big(\; & \texttt{ChapmanMeanLength}[\text{title}] \geqslant 0.978 \\
& \bigwedge \neg \big(\texttt{OverlapGram}_3[\text{author}] \geqslant 0.411 \big) \\
& \bigwedge \neg \big(\texttt{MongeElkan}[\text{volume}] \geqslant 0.429 \big) \big) \bigvee \\
\big(\; & \texttt{CosineGram}_2[\text{title}] \geqslant 0.730 \\
& \bigwedge \texttt{OverlapGram}_3[\text{author}] \geqslant 0.411
\end{aligned}
$$

$\bigwedge \neg \left(\texttt{MongeElkan}[\mathsf{volume}] \geqslant 0.429\right)\big)$

Here, we present one representative **GBF** detected by RuleSynth (or its variants) for each of the remaining datasets. φ_{synth} indicates a **GBF** generated using RuleSynth.

- Amazon-GoogleProducts:

$$\varphi_{synth}: \ \mathsf{CosineToken}[\mathsf{title}] \geqslant 0.571$$

- Locu-Foursquare:

$$\begin{aligned}\varphi_{synth}: \ & \textbf{if} \quad \mathsf{noNulls}[\mathsf{region}] \geqslant 1 \\ & \textbf{then} \quad \big(\big(\mathsf{MongeElkan}[\mathsf{street\_address}] \geqslant 0.861 \\ & \qquad \bigvee \big(\mathsf{ChapmanMeanLength}[\mathsf{locality}] \geqslant 0.161 \\ & \qquad\quad \bigwedge \mathsf{Jaro}[\mathsf{name}] \geqslant 0.753\big)\big)\big) \\ & \textbf{else} \quad \mathsf{EuclideanDistance}[\mathsf{name}] \geqslant 0.678\end{aligned}$$

- DBLP-Scholar:

$$\begin{aligned}\varphi_{synth}: \ & \big(\big(\mathsf{SmithWatermanGotoh}[\mathsf{title}] \geqslant 0.84 \\ & \quad \bigwedge \ \textbf{if} \quad \mathsf{noNulls}[\mathsf{year}] \geqslant 1 \\ & \qquad\quad \textbf{then} \quad \mathsf{OverlapGram}_2[\mathsf{year}] \geqslant 0.833 \\ & \qquad\quad \textbf{else} \quad \mathsf{MongeElkan}[\mathsf{authors}] \geqslant 0.9\big) \\ & \quad \bigvee \big(\mathsf{ChapmanMatchingSoundex}[\mathsf{title}] \geqslant 0.965 \\ & \qquad \bigvee \mathsf{NeedlemanWunch}[\mathsf{title}] \geqslant 0.807\big) \\ & \quad \bigwedge \big(\mathsf{ChapmanMeanLength}[\mathsf{authors}] \geqslant 0.536 \\ & \qquad \bigvee \mathsf{SmithWaterman}[\mathsf{authors}] \geqslant 0.846\big)\big)\end{aligned}$$

As described earlier (Sec. 4.2), RS-SynthComp and RS-Consensus produce larger **GBF**s by assembling smaller **GBF**s. $\varphi_{SynthComp}$ is one such composite **GBF** generated by RS-SynthComp, and $\varphi_{Consensus}$ is another such composite **GBF** produced by RS-Consensus for the Amazon-GoogleProducts dataset.

$\varphi_{SynthComp}$: $\big((\mathsf{a0} \bigvee \mathsf{a1}) \bigwedge \mathsf{a2}\big)$

$\mathsf{a0}$: $\mathsf{CosineToken[title]} \geqslant 0.55$

$\mathsf{a1}$: $\mathsf{OverlapGram_3[title]} \geqslant 0.609$

$\mathsf{a2}$: $\mathsf{CosineToken[title]} \geqslant 0.489$

$\varphi_{Consensus}$: $\texttt{count\_true}\big(\mathsf{a0}, \mathsf{a1}, \mathsf{a2}, \mathsf{a3}, \mathsf{a4}\big) \geqslant 3$

$\mathsf{a0}$: $\mathsf{EditDistance[title]} \geqslant 0.758 \bigvee \mathsf{OverlapToken[title]} \geqslant 0.817$

$\mathsf{a1}$: $\mathsf{CosineToken[title]} \geqslant 0.309 \bigwedge$
$\big(\mathsf{EditDistance[manufacturer]} \geqslant 0.550 \bigvee \mathsf{OverlapGram_2[title]} \geqslant 0.861\big)$

$\mathsf{a2}$: $\mathsf{CosineToken[title]} \geqslant 0.601$

$\mathsf{a3}$: **if** $\big(\mathsf{noNulls[manufacturer]} \geqslant 1\big)$
then $\mathsf{SmithWaterman[description]} \geqslant 0.178$
else $\mathsf{OverlapToken[title]} \geqslant 0.55$

$\mathsf{a4}$: **if** $\big(\mathsf{noNulls[manufacturer]} \geqslant 1\big)$
then $\mathsf{Soundex[title]} \geqslant 0.978$
else $\mathsf{CosineGram_2[title]} \geqslant 0.646$

Figure 8-5 shows the interpretability results with respect to the number of atoms for all datasets. It shows that the RuleSynth and the RS-SynthComp algorithms produce more interpretable rules, i.e., with fewer atoms, than decision trees with depths 3 and 4 for all datasets. In particular, RuleSynth produces rules that are (i) more interpretable than decision trees with depth 3 for all datasets and (ii) up to eight times more interpretable than decision trees with depth 4 (see dataset `Amazon-GoogleProducts`). The RS-SynthComp algorithm produces rules with more atoms but still has better interpretability than decision trees with depth 4. Moreover, as we will see in Exp-2, the rules produced by RS-SynthComp are more effective than decision trees with both depth 3 and 4. The RS-Consensus algorithm produces rules with even more atoms, but they still have fewer atoms than *small* gradient tree boosting and random forests. Small gradient tree boosting and random forests are also not easily interpretable due to the presence of numerical weights along with the small trees.

Figure 8-5 also tells us that the number of atoms increases exponentially with the depth of the decision trees i.e., the deeper is the tree, the less interpretable the corresponding rules are. For example, it is nearly impossible to interpret decision trees of depth 10 with

thousands of atoms.

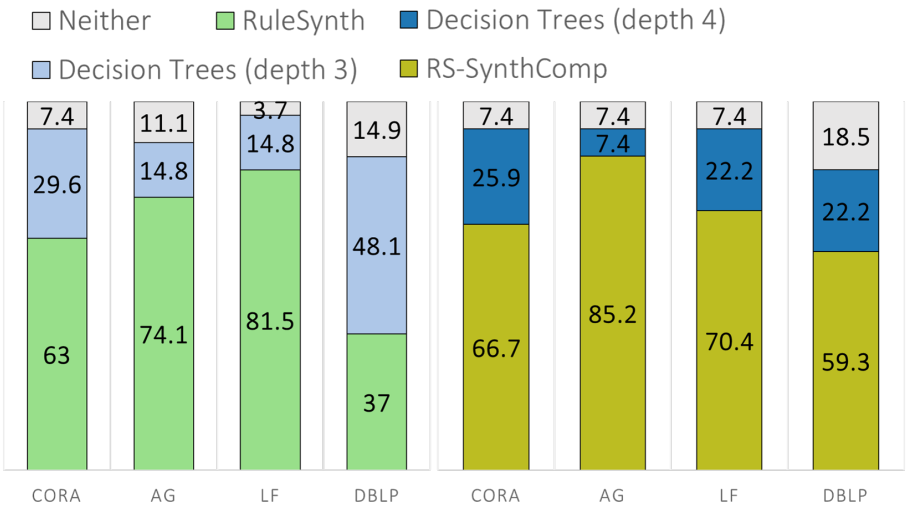

Figure 8-6: User interpretability preference: Cora, Amazon-GoogleProducts (AGP), Locu-FourSquare (LFS), DBLP-Scholar (DBLP)

User study. Figure 8-6 shows the results of our informal user study with 27 CS researchers from six institutions. We gave each participant 8 multiple-choice questions with 3 options for the answer. Each question comprised a pair of well-formatted rules generated by two different techniques from the same training data. The participant was asked to select *which one of the two rules they thought was more interpretable* (an example question and rules are shown in Fig. 8-8). Each participant was given 2 questions for each dataset. One question compared the rules generated by RuleSynth against decision trees of depth 3, and the other compared RS-SynthComp against decision trees of depth 4. We observe from the results that the rules with fewer atoms are preferred by more users. Generally the rules generated by RuleSynth and its variants are preferred except in one case (i.e., RuleSynth on `DBLP-Scholar`), where the decision trees have a similar number of atoms as our algorithms, as shown in Figure 8-5. On average, 67.15% of the responses state that the rules generated by our algorithms are more interpretable, while only 23.13% prefer the decision trees, and 9.72% state no preference. This supports the validity of #-atoms as our measure of interpretability.

These results overall showcase the ability of EM-Synth to generate concise **GBF**s, yielding compact and more interpretable rules than other interpretable methods.

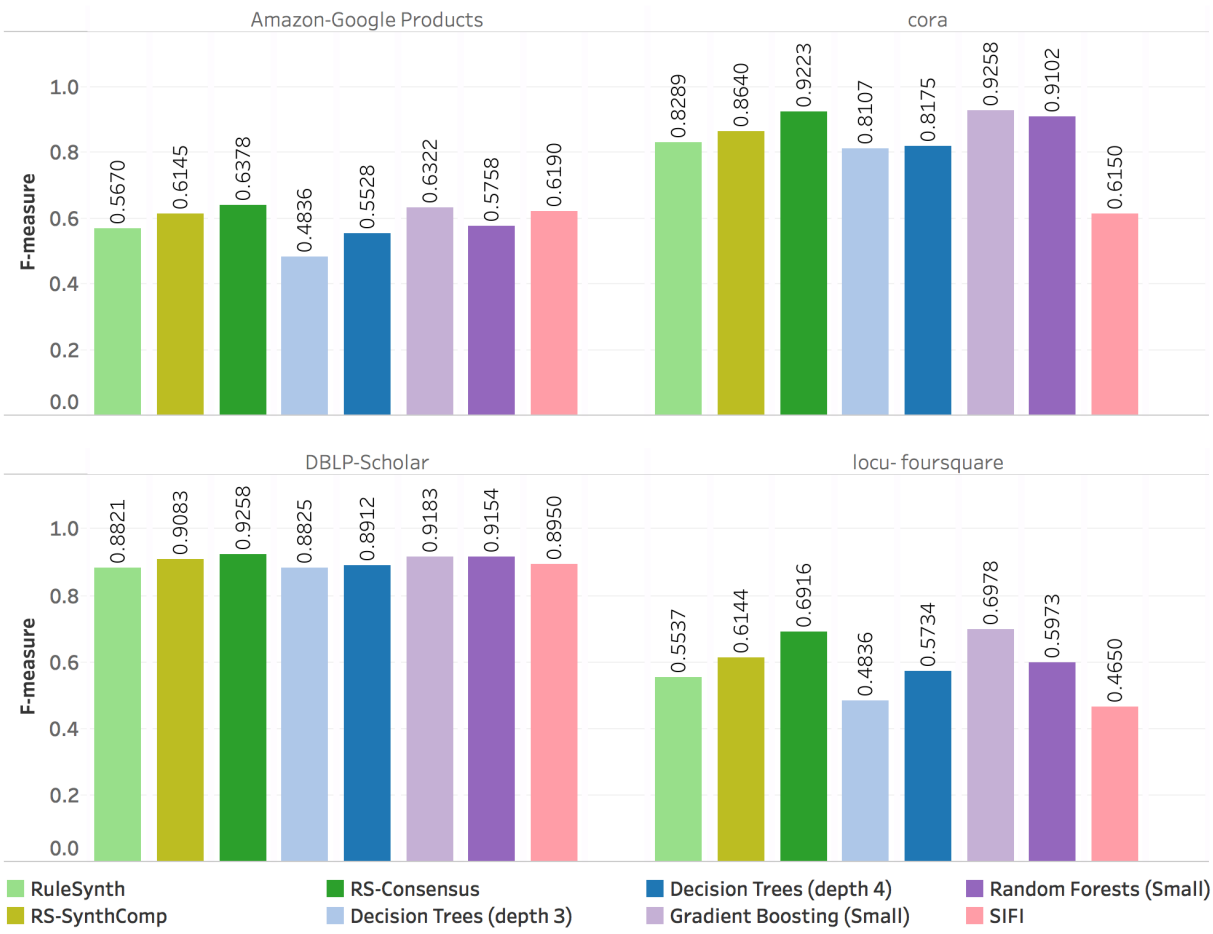

Figure 8-7: Effectiveness results for 5-folds experiment (80% training and 20% testing data)

8.4.2 Exp-2: Effectiveness vs. interpretable decision trees

We now evaluate the effectiveness of rules generated by our algorithms against the ones found by decision trees. As mentioned before, we use the average F-measure across 5 folds as the effectiveness metric.

Figure 8-7 shows the average F-measures for different interpretable techniques. We observe that RuleSynth achieves a higher F-measure than decision trees with depth 3 for all datasets, except for `DBLP-Scholar` where the F-measures are comparable. Decision trees achieve higher F-measures when increasing their depth from 3 to 4 for all datasets. However, RS-SynthComp still results in higher F-measures than decision trees with depth 4 on all data sets.

Moreover, as we saw in Figure 8-5, each of RuleSynth and RS-SynthComp produces more interpretable rules than decision trees with depth 4 for all datasets. From Figures 8-5

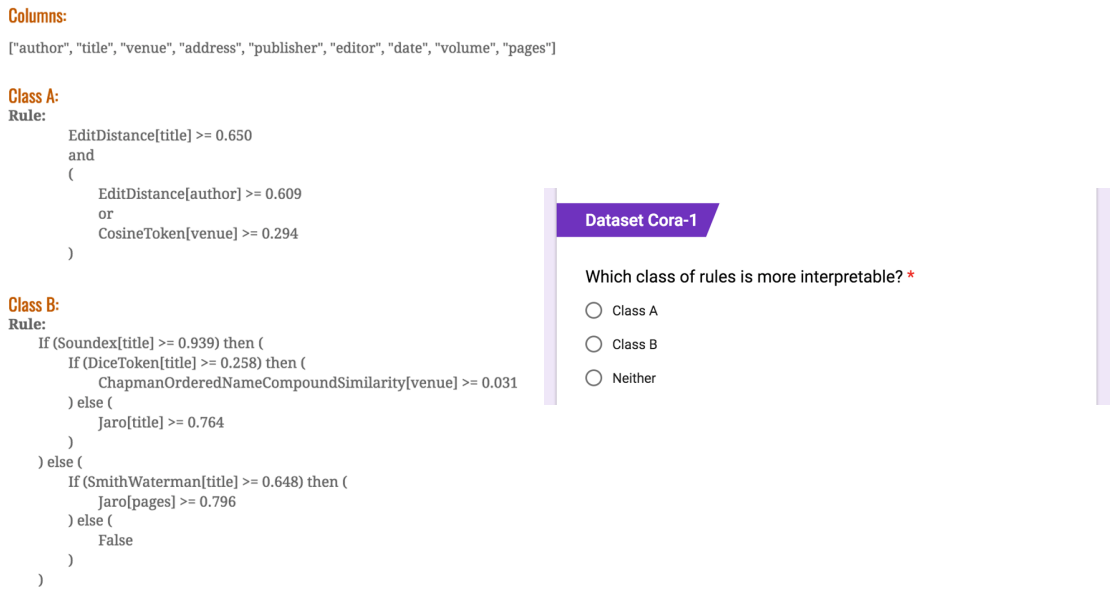

Figure 8-8: Interpretability user study form for the participants

and 8-7, we conclude that decision trees can get better F-measures by increasing their depth, but this comes at a significant sacrifice to their interpretability. In contrast, EM-Synth can get better F-measures by using the RS-SynthComp and RS-Consensus algorithms while not sacrificing interpretability as much. For example, for the `Amazon-GoogleProducts` dataset, increasing the depth of decision tree from 3 to 4 increases the F-measure from 0.484 to 0.553 while the average number of atoms increases from 4.6 to 10.2. In contrast, the RS-SynthComp increases the F-measure from 0.567 to 0.614 while increasing the average number of atoms from 1.4 to 4.2.

8.4.3 Exp-3: Effectiveness vs. expert-provided rules

To further demonstrate the effectiveness of **GBF**s produced by our algorithms, we compare our algorithms with SIFI [117]. SIFI requires experts to provide a **DNF** template from experts as an input and completes it to generate a rule. In contrast, algorithms from EM-Synth discover rules automatically, reducing the effort needed from an expert.

Figure 8-7 shows that algorithms from EM-Synth perform better than SIFI for all datasets. In contrast with SIFI, which employs a heuristic to search through a smaller space of rules, RuleSynth searches through a huge space of generic **GBF**s. This allows us to discover various corner cases that can be sometimes missed by an expert-provided ex-

pression. In addition, as shown in Figure 8-5, RuleSynth generates **GBF**s that are more concise (and thus interpretable) than the **DNF**s produced by SIFI for all datasets.

8.4.4 Exp-4: Effectiveness vs. non-interpretable methods

We now compare the algorithms from EM-Synth with four ML algorithms: (1) Decision trees with depth 10, (2) SVM, (3) Random forests, and (4) Gradient tree boosting. Figure 8-7 shows the results for interpretable methods, and Figure 8-9 gives the results for non-interpretable methods. We observe that all three algorithms RuleSynth, RS-SynthComp, and RS-Consensus achieve smaller F-measure values than the ML algorithms on an average. Still, RS-Consensus achieves quite comparable F-measures, with the F-measure difference between the ML best algorithm and RS-Consensus being 0.08, 0.05, 0.02, and 0.04 for each of the four data sets. However, the effectiveness of these ML algorithms comes at a high price. We see in Figure 8-5 that these four ML algorithms are not interpretable: (i) SVM does not produce rules, (ii) decision trees with depth 10 yield rules with around $1K$ atoms for all datasets, (iii) random forests and gradient tree boosting provide both rules with 1K-13K atoms with hundreds of weights, which are also impossible to interpret for a human.

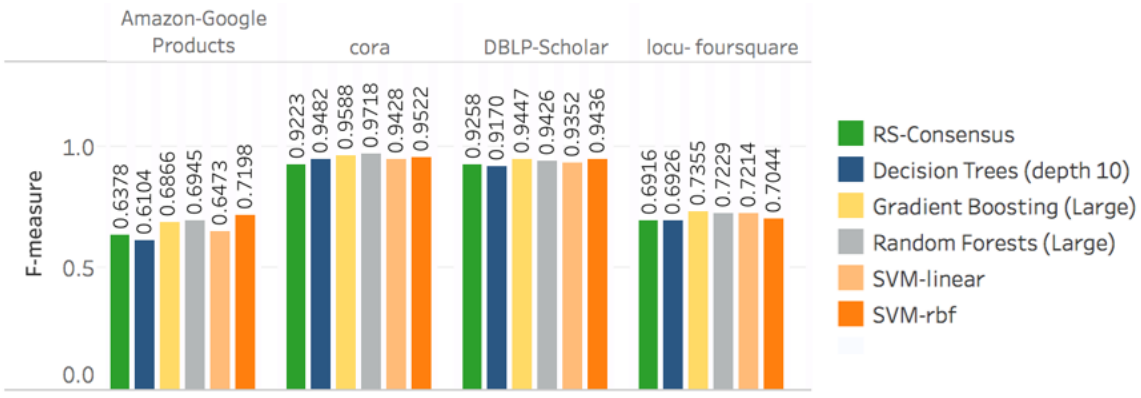

Figure 8-9: Effectiveness results for 5-folds experiment: RS-Consensus vs. non-interpretable methods

8.4.5 Exp-5: Variable training data

In this experiment, we vary the default number of folds ($K = 5$) by randomly sampling a fraction $\frac{1}{K}$ of training examples with $K = 100, 40, 20, 10, 7, 5$. Each fraction $\frac{1}{K}$ corresponds

to a different percentage $P\%$ of examples (i.e., $P = 1, 2.5, 5, 10, 14.3, 20$). We use the rest $\big((100 - P)\%\big)$ of the examples for testing, and we train and test on 100 such randomly selected sets for each percentage P. We report the average test-set F-measure and size of matching rules obtained across all 100 runs (with 99% confidence intervals) for each dataset.

We used a cluster with 70 parallelism to run these experiments. For each of the 4 datatsets, 4 techniques and 6 training-data percentages – we run the experiment 100 times on different random training sets (note that the training sets were kept the same across each technique). These $4 \times 4 \times 6 \times 100 = 9600$ runs with 70 parallelism took 36 hours in total with various techniques taking different times on average across their 3000 runs each: (1) RuleSynth: 2436 s, (2) RS-SynthComp: 663 s, (3) Decision trees (depth 3): 79 s, (4) Decision trees (depth 4): 86 s. Figure 8-10 shows the comparison between interpretable decision trees and algorithms from EM-Synth on the Locu-Foursquare dataset with different percentages (1% to 20%) of training data. The figures for the other datasets show similar trends in Figures 8-11,8-12,8-13.

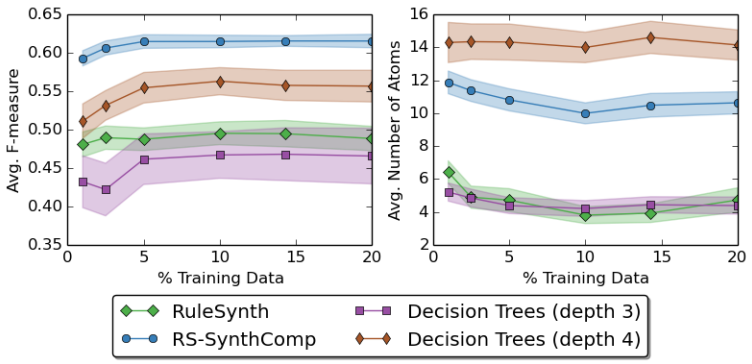

Figure 8-10: Locu-Foursquare (100 runs with 99% CIs on the means in the shaded regions)

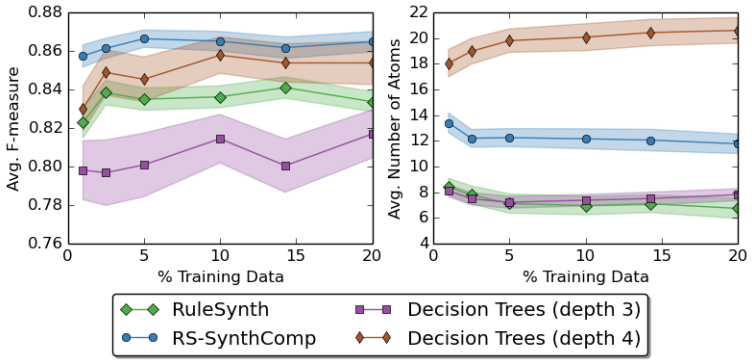

Figure 8-11: Cora (100 runs with 99% CIs on the means in the shaded regions)

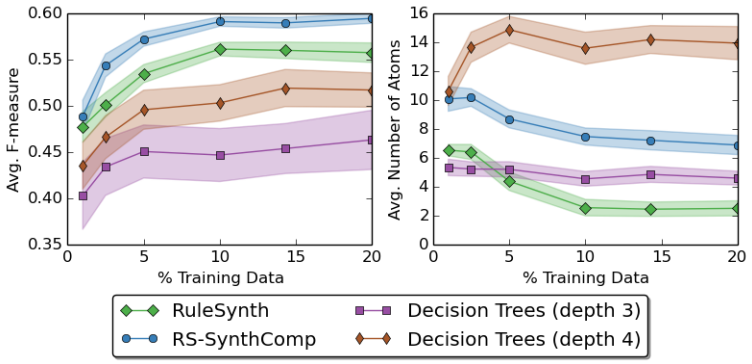

Figure 8-12: Amazon-GoogleProducts (100 runs with 99% CIs on the means in the shaded regions)

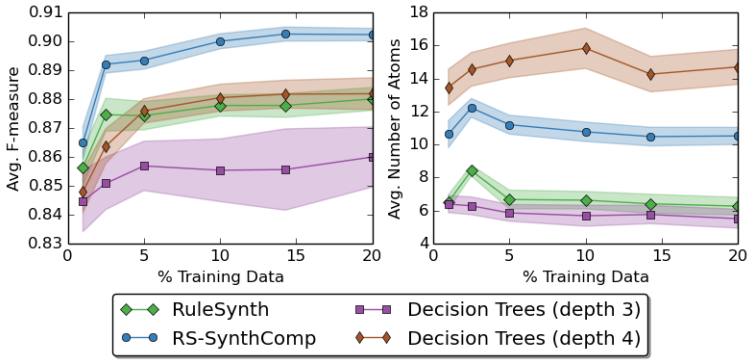

Figure 8-13: DBLP-Scholar (100 runs with 99% CIs on the means in the shaded regions)

We compare decision trees (depth 3) with RuleSynth since they both produce rules with smaller sizes, and decision trees (depth 4) with RS-SynthComp since they both produce interpretable rules with larger sizes. Both RuleSynth and RS-SynthComp outperform decision trees of depth 3 and 4, respectively, in effectiveness (higher F-measure) on all datasets. At the same time, RS-SynthComp generates more interpretable (lower number of atoms) rules than decision trees (depth 4). RuleSynth and decision trees (depth 3) both generate small and interpretable rules (2-7 atoms on average). RuleSynth generates smaller rules for 3 out of 5 datasets, has similar interpretability for `Locu-Foursquare`, and generates slightly larger rules for `DBLP-Scholar`. RS-SynthComp is the most effective method for generating interpretable rules (2-14 atoms on average) with limited training data on all datasets.

8.4.6 Exp-6: Efficiency of training

The algorithms from EM-Synth provide the flexibility for users to control how much the algorithm should explore in the CEGIS loop (bound K_{CEGIS}, time limit, grammar bounds) and how many times it should restart (bound K_{RANSAC}). Figure 8-14 shows that the algorithms from EM-Synth take at most an hour to search through the huge space of rules in order to produce an effective and concise rule as output for all datasets in Fig. 8-3. This is a reasonable amount of time as compared to what it takes experts to examine the dataset and write their own rule expressions, especially given the low cost of computation relative to human time. For example, our experts took around 2 hours on average to write a **DNF** expression for SIFI per dataset.

Figure 8-14: Efficiency of training (average time for training per fold) for 5-folds experiment (80% training / 20% testing)

Figure 8-14 also shows that SIFI searches through a smaller constrained space in at most 40 minutes to produce a rule. Decision trees with depth 3 and 4 produce a rule in less than a minute, but the produced rules are neither as concise nor as effective as rules produced by

RuleSynth and its variants (Exp-2). Both decision trees with depth 10 and SVM take at most 8 minutes to produce a rule, but they are not designed to expose interpretable results (Exp-4).

8.4.7 Exp-7: Efficiency of Testing

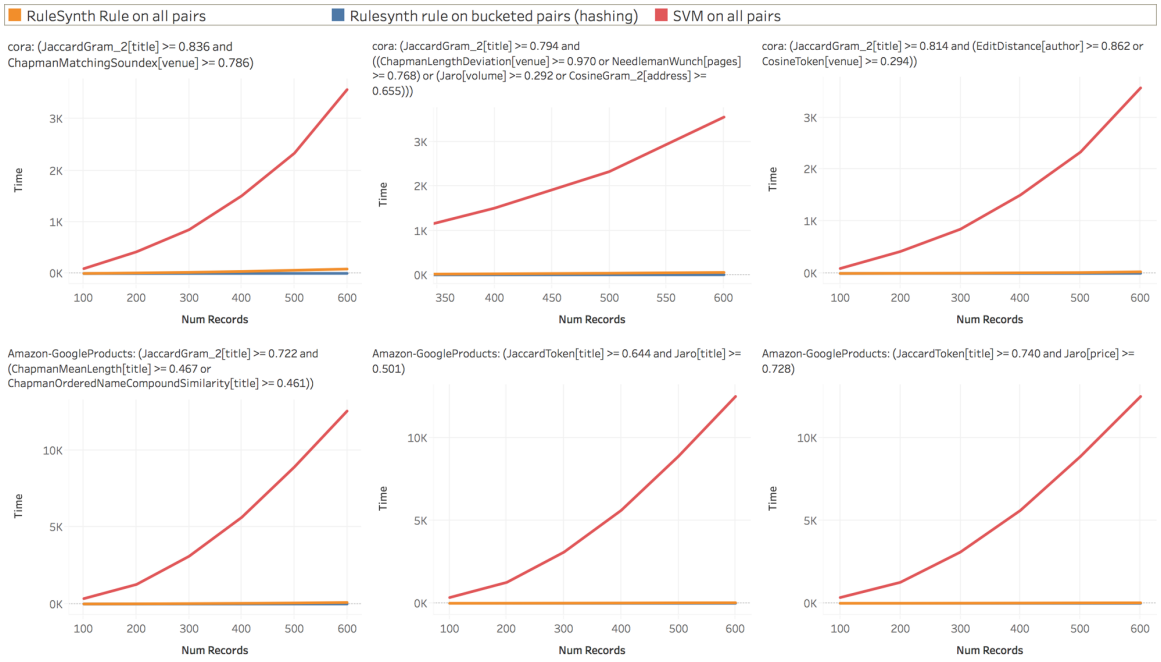

Figure 8-15: Efficiency of testing a classifier: SVM vs. RuleSynth-generated rule on all pairs and bucketed pairs

When we have N records and we want to apply a rule or a classifier on each pair of these records to identify duplicates, in general, applying a rule would require enumerating all $O(N^2)$ pairs and computing the relevant similarity functions. Note that until now, for training, we pre-computed these similarity functions, but now, for testing the rule in a new environment, we have to compute the relevant similarity functions again to be able to apply the rule or the classifier. For a classifier that uses many similarity functions, this process becomes prohibitively slow since they have to compute all of them; as shown in Fig. 8-17, SVM is much slower than rules applied on all pairs. Hence, applying smaller rules on all pairs already has an advantage over large classifiers that utilize many similarity functions with numerical weights like SVM. Moreover, for smaller **GBF** rules with a specific structure, one can use a hashing scheme [118] to bucket similar records as a first pass, which reduces the pairwise comparisons from $O(N^2)$ to $O(M^2)$ where $M \ll N$ (as explained in Subsec. 8.1.4).

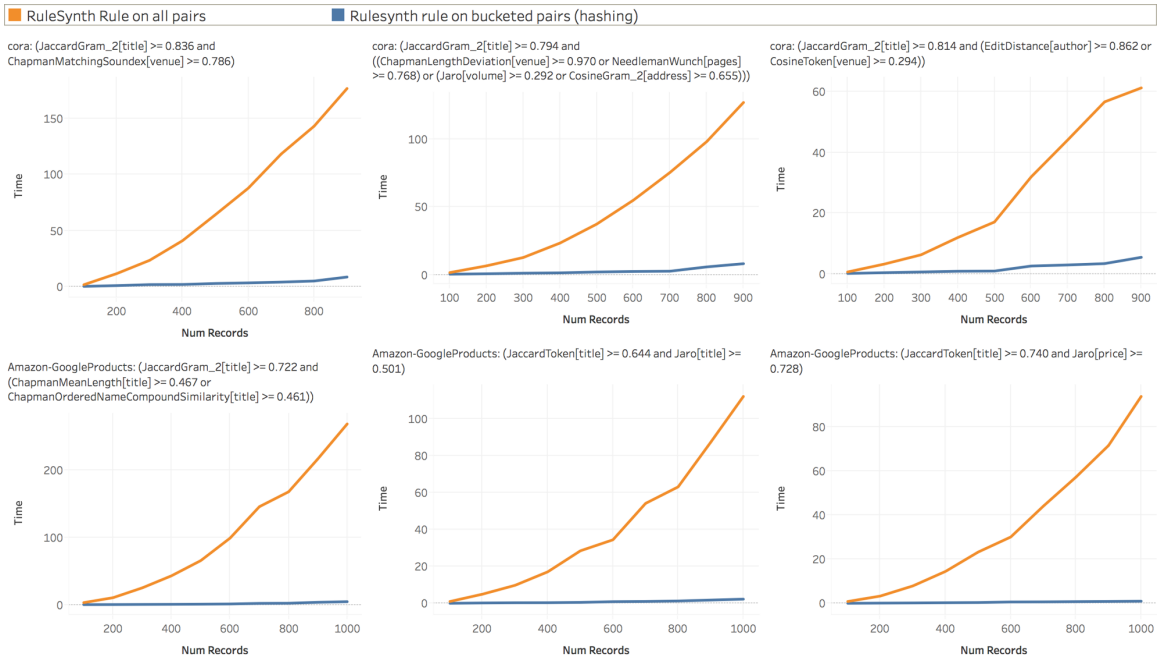

Figure 8-16: Efficiency of testing a classifier: RuleSynth-generated rule on all pairs vs. bucketed pairs

The rule application on buckets takes much less time $(1-4\text{s})$ than applying the rule on all pairs $(30-200\text{s})$, as shown in Fig. 8-17. The full variation of time taken for testing against the number of records used for the rules in Fig. 8-17 can be seen in Figures 8-15 and 8-16.

	rule	**# record pairs**	**time taken (s)**		
			SVM	**rule on all pairs**	**rule on buckets**
$\mathsf{D_C}$	φ_1	360,000	3,576.18	55.17	2.56
	φ_2			32.05	2.67
	φ_3			87.91	3.63
	φ_1	810,000	N/A	127.29	8.30
	φ_2			61.38	5.54
	φ_3			176.46	8.85
$\mathsf{D_{AG}}$	φ'_1	360,000	12,528.3	30.28	0.80
	φ'_2			34.47	1.02
	φ'_3			98.94	1.17
	φ'_1	810,000	N/A	71.80	1.03
	φ'_2			87.23	1.9
	φ'_3			216.61	3.65
$\mathsf{D_C}$ = **Cora**, $\mathsf{D_{AG}}$ = **Amazon-GoogleProducts**					

Figure 8-17: Efficiency of testing

We identified 3 RuleSynth-generated rules each for two datasets that are of the form $(\mathsf{f_{sim}}[\mathsf{attr}] \geqslant) \theta \wedge \varphi'$ where φ' is a general **GBF** and f_{sim} is a similarity function for which there is a locality-sensitive-hashing (LSH) family available [118]. Note that in EM-Synth, we can also force this structure for all rules with our flexible grammar. For this experiment, we found rules generated by RuleSynth that have the format mentioned above with f_{sim} being the Jaccard function over the set of n-grams (with $n = 2$) of the input strings. Using the MinHash LSH scheme described in Subsec. 8.1.4, we built an index on the attribute attr with Jaccard threshold θ to identify potentially similar pairs and reduce the number of record pairs to compare.

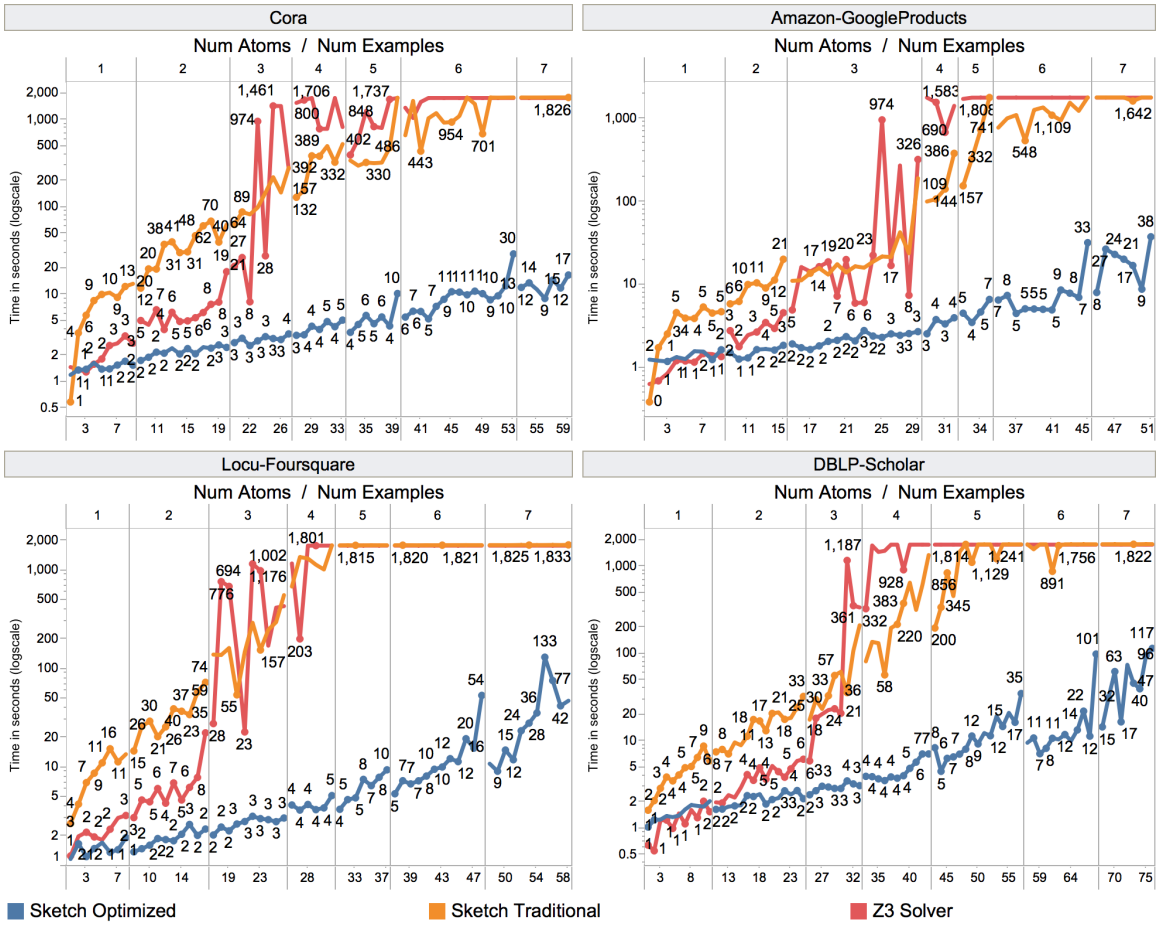

Figure 8-18: Time taken by traditional Sketch, Z3 [41] solver and Sketch with custom synthesizer (Sketch Optimized).

8.4.8 Exp-8: Impact of the custom synthesizer in SKETCH

In this subsection, we show that using the special-purpose synthesizer (Subsec. 2.3.3) in SKETCH is very important for EM-Synth to scale well. To show this we compare the running times of the different solvers on the synthesis instances from one CEGIS loop (multiple iterations with increasing number of examples) for all 4 datasets. The comparison is presented in Fig. 8-18. Note that the grammar bound N_a (denoted by "Num Atoms" in the figure) is incrementally increased from 1 to 7, and there are discontinuous jumps in the graphs when the value of N_a changes, because it implies that there was no rule for $N_a = i$ with n examples, so the EM-Synth system tries to find a rule with $N_a = i + 1$ again with the same n examples. As we can see from Fig. 8-18, the running times for many synthesis instances was decreased from more than 30 minutes (timeout for these runs) to less than a minute and in most cases less than 20 seconds when using the custom synthesizer. Without this special-purpose synthesizer, the overall time for even one CEGIS loop will go up to a day and this may make the rule discovery prohibitively slow for practical purposes.

Chapter 9

Related Work

In this section we discuss the relevant related work. We start with the related work for the overall framework (Sec. 9.1) and then discuss the related work for each of the systems instantiated from this framework i.e., the SWAPPER system (Sec. 9.2) and the EM-Synth system (Sec. 9.3).

9.1 Overall framework

9.1.1 Combining program synthesis and machine learning

The area of combining statistical machine learning with program synthesis has seen a renewed interest and there has been a lot of recent work that aims to use statistical learning for program synthesis. Recent tools built using the programming by example [56] framework, target *data wrangling* problems such as data transformation, extraction and cleaning. These tools efficiently synthesize desired programs with a probabilistic or numerical metric being optimized [95, 99]. The key idea in these tools is to first learn all the programs consistent with a given set of examples (represented succinctly using version-space algebra), and then rank them using a ranking function to return the most desirable program amongst them. The ranking function can either be provided by a domain expert or can be learned using machine learning from training benchmarks [98].

Another recent work [90] provides a way to use statistical optimization techniques to synthesize programs from supervised example-based I/O data that can also potentially tolerate some noise in the data. The framework presented in [90] can be seen as a special case

of our framework for supervised learning problems with no logical correctness constraints on the function to be synthesized. This approach was used to synthesize bitstream programs from a dataset of examples with some noise, and programs in a domain-specific language (DSL) that condition the predictions made by a statistical code completion system.

There has also been a lot of recent interest in using deep learning for program synthesis [101]. The key idea in these approaches is to design neural architectures that can encode a specification (such as input-output examples or natural language) and generate the corresponding program as the output. Since the neural architectures are trained on a large amount of training programs (typically generated synthetically) in a given domain, the networks are able to synthesize new programs in the same domain from very few input-output examples. This approach has been used to learn string transformation programs in the RobustFill system [43]. The RobustFill system uses LSTM networks and encodes input and output strings as a sequence of characters, whose hidden representations are then fed to an LSTM sequence decoder to generate tokens in the DSL as an output program. Another approach uses Reverse-Recursive Recurrent Neural Networks (R3NNs) [84] as decoders in comparison to sequence decoders that tries to also encode the tree structure of the programs. A similar approach based on R3NNs has also been applied to learn composition of API functions that denote semantic string transformations [20]. Neural-networks-based methods have also been used to synthesize grammars from example input files to enable grammar-based fuzzing [53] and automatically learning error models (a set of rewrite rules) for synthesizing repairs for student submissions to introductory programming assignments [19, 89].

9.1.2 Program synthesis with quantitative objectives

Quantitative program synthesis for reactive systems is another active area of research where programs (or reactive controllers) are constructed from specifications provided as automata for correctness of its behavior along with some quantitative objective [23] to be optimized. This objective is usually computed over many possible program behaviors e.g., worst-case costs, expected rewards over a probability distribution [32], expected rewards with partial information [30] etc. These synthesis techniques have been applied to different domains like robust or self-fixing programs [22], probabilistic systems [32], concurrent data structures [30]. Unlike our framework, these techniques rely on analyzing the behaviors for correctness and rewards at the same time. Moreover, the numerical objective (or score) is limited to a few

options typically studied in the corresponding theory of games or Markov decision processes that are used to solve the synthesis problems [113].

9.1.3 Synthesis of components

The sub-problem of synthesizing components having logical structure provided by a grammar is solved by using application-domain-specific knowledge. This is analogous to doing synthesis in domain-specific languages (DSLs). In our framework, the synthesis-of-components step corresponds to doing synthesis in the DSL of the components being considered.

In recent years, program synthesis has seen a wide variety of applications when targeted to specific domains e.g., [42, 52, 57, 88, 114, 119]. Domain specificity narrows the search space for programs and makes it possible to bake domain-specific insights into the synthesis algorithm to make it efficient. That being said, a lot of these applications have been one of a kind i.e., for every DSL, researchers develop their own search procedures, custom data-structures and generally base their algorithms on explicit enumeration with some domain-specific pruning.

In this thesis, we describe two ways of incorporating domain specific insights while performing synthesis with a general purpose synthesizer called SKETCH [111]: (1) an outer refinement method to maintain high-level domain-specific constraints while querying SKETCH repeatedly and prune the high-level constraints based on the feedback from SKETCH (2) an inner domain-specific theory solver that can be fitted inside the SKETCH synthesizer to solve problems from these domains efficiently. This customization of a general purpose solver (SKETCH) enables reusing a lot of infrastructure already built in SKETCH for our domain-specific synthesis problems.

9.2 SWAPPER system

A recent paper introducing Alive [76], a domain specific language for specifying, verifying, and compiling peephole optimizations in LLVM is the closest to SWAPPER 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.

9.2.1 Formula rewriting in constrain solvers

A pre-processing step in constraint solvers and solver-based tools (like Z3, Boolector [27], SKETCH etc) is an essential one and term rewriting has been extensively used as a part this pre-processing step [28, 31, 36, 73]. These pre-processing steps are very important and can have a significant impact on performance.

9.2.2 Pattern finding

In the context of Motif discovery problem [92] (finding recurrent sub-graphs), recently we have seen some attempts to use machine learning [67] and distributed algorithms [75] to compute the Motifs efficiently. 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.

9.2.3 Comparison with superoptimization

In the superoptimization community, researchers have been working on building compilers that 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 [14] 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 SWAPPER, there are a few differences. Most importantly, [14] 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. Recently, there has also been some work [87] that uses synthesis and other stochastic search techniques to help the researchers generate a superoptimizer efficiently for different instruction set architectures.

9.2.4 Code generation

Code generation phase in SWAPPER that automatically generates simplifier's code, is similar to a term or graph rewrite system like Stratego/XT [24] or GrGEN.NET [51]. 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 (the SKETCH simplifier). Our strategy is comparable with LALR parser generation [58] where the next look-ahead symbol helps decide which rule to use.

9.3 EM-Synth system

9.3.1 Machine Learning-Based Entity Matching

Most current EM solutions are variants of the Fellegi-Sunter model [47], where entity matching is treated as a *classification* problem: given a vector of similarity scores between the attributes of two entities, the problem is to determine whether two entities are *matching* or *non-matching*. Such approaches include SVM-based methods [21], decision-tree-based solutions [33, 54], clustering-based techniques [39, 93], and Markov-logic-based models [107].

Most machine learning models, except shallow decision trees, are hard to interpret. EM-Synth produces declarative rules that are often preferred by the end users. Another drawback, as pointed out by [117], is that these methods are rather expensive due to the quadratic complexity of tuple pair enumeration, even with the help of blocking [16] and canopy filtering [77].

9.3.2 Rule-based entity matching

Declarative entity matching rules are highly desirable by the end users, since users can understand how entities are matched in a deterministic way. Such rules are also popular in the database community since they provide great opportunities for improving the performance at the execution time, such as those studied in [17, 40, 66]. However, these approaches typically assume that the entity matching rules have been defined by domain experts, which in practice, are hard to come up with.

Closer to our work is the SIFI tool [117], that automatically discovers similarity functions and their associated thresholds by assuming that the rule is given as a **DNF**. As compared to SIFI, our approach can automatically discover the optimal and more expressive **GBF** based rules with corresponding similarity functions and thresholds without any user input i.e., we provide an end-to-end solution for generating entity matching rules. Another recent work [83] proposes an unsupervised learning method for link discovery configuration for the Web of Data. Their solution is based on three assumptions: (1) no duplicate records exist within each dataset in terms of URIs; (2) two datasets have a strong degree of overlap; and (3) a meaningful similarity function returns values close to 1.0 for matching pairs. They rely on the above three assumptions to compute indicators of "good characteristics", i.e., to simulate user examples. Although the solution works well for the Web of Data, the above three assumptions do not hold for more general entity matching settings, as studied in this work.

9.3.3 Active learning and crowdsourcing

Since good and sufficient training dataset is always hard to get in practice, a natural line of study deals with actively involving users in verifying ambiguous tuple pairs, *a.k.a.* active learning in entity matching [54, 81, 93]. Also, due to the popularity of crowdsourcing platforms, there have been efforts to leverage crowd workers for entity matching problems [48, 54, 115].

We assume that training data is already given as an input and focus on using EM-Synth to synthesize EM rules as a solution of the studied problem. Hence, the above works are essentially orthogonal, but complementary, to the rule-synthesis problem studied in this work. In other words, if the provided training dataset is not good enough, we can use their techniques to interact with users or crowd workers for getting more and better training data.

9.3.4 Program synthesis for databases

Recently, program synthesis and programming by example has shown great promise for database related problems, such as data transformation [96], social recommendations [38], and translating imperative code into relational queries [35]. Similar to the above mentioned works, in EM-Synth, we showcase the potential of program synthesis in supporting another important problem in data integration and data cleaning – generating entity-matching rules.

9.3.5 Special-purpose constraint solvers

Using a custom solver inside SKETCH is similar to using a special purpose theory solver [50] inside a different class of solvers, namely, the SMT solvers. In the context of program synthesis, we are the first to show how a custom solver can be used to solve synthesis problems efficiently inside a general purpose solver like SKETCH.

Chapter 10

Conclusion

We conclude this thesis by reiterating that this work aims to push the boundaries between program synthesis and machine learning (ML). We are able to solve certain problems that couldn't be solved easily or efficiently using traditional techniques. We presented a general framework that has been applied to multiple domains successfully, including the two that were discussed in depth in this thesis: (1) formula simplification in constraint solvers (Swapper system) and (2) database entity matching with concise and interpretable rules (EM-Synth system). This work has also enabled new strategies for adapting a general purpose synthesizer (like Sketch) to domain specific synthesis problems, thereby helping the developers who can reuse the infrastructure already present in the general purpose solvers. This work is aimed towards making the developer's life easy. With the Swapper system we are automating software engineering for simplifiers, and with EM-Synth we are automatically providing good interpretable EM rules to database experts. This goal is at the heart of both program synthesis and machine learning, and we believe that we can do more to satisfy this goal if we push boundaries between the two fields further, as we did in this work.

Appendix A

Synthesis with SKETCH: implementation notes

In this chapter, we present some implementation notes on how one can use SKETCH to solve the relevant parts of the synthesis-of-components problems discussed in this thesis. We start by introducing the SKETCH system and the required notation to present the implementation details.

A.1 SKETCH synthesis system

SKETCH is an open-source system for synthesis from *partial programs*. A *partial program* represents a space of possible programs by giving the synthesizer explicit choices about what code fragments to use in different places. More precisely, a partial program can be seen as a C-like 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 or constraints, the SKETCH synthesizer finds a completion for the holes that satisfies the constraints for all inputs from a given input space. SKETCH uses symbolic execution to derive a formula $P(x, c)$ 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. $\exists c\ \forall x\ P(x, c)$.

Note that this formulation can be used to represent all SyGuS problems as defined earlier in Sec. 2.1. The example SyGuS problem from Sec. 2.1 can be represented as a partial program as shown below:

```
void tester(bit x, bit y) {
    int bnd = 0;
    bit t = boolExp(x, y, bnd);
    assert bnd <= B;
    if (x) assert t ==~y;
    if (y) assert t == ~x;
}

generator bit boolExp(bit x, bit y, ref int bnd){
    if (??){
        bnd++;
        return boolExp(x,y) || boolExp(x,y);
    }
    else{
        return {| x | y | ~x | ~y |};
    }
}
```

The partial program above (also called a sketch) gives the synthesizer a space of possible code fragments in three ways:

1. A regular expression generator `{| regexp |}` where the regexp can use the operator | to describe choices among expressions. SKETCH also supports choices among operators in a similar manner, for example: `{| x ( + | * | - ) y |}` represents the space of applying any of those 3 operators between variables x and y.

2. A constant generator or a *hole* represented by `??` that takes integer values from a bounded set. In the case when this hole is inside the condition of an `if` statement, it will choose a value between 0 (false) or 1 (true).

3. Recursive generator functions that recursively define program spaces (e.g., the function `boolExp` in the example above). Any recursive generator function will be inlined with new holes and regular expression generators inside it for every use of this function.

Moreover, to enforce the bound from the grammar, we use a variable *bnd* that counts how many times the relevant production rule of the grammar has been applied. This variable *bnd* is passed by reference (denoted by the `ref` keyword in SKETCH) and asserted to be less than or equal to the bound B.

In summary, in the sketch above, the synthesizer can choose whether or not to negate x and y before or-ing them together in order to satisfy the assertions in the tester. More

details on the syntax of the SKETCH language and the best practices can be found in the SKETCH manual [6].

A.2 SWAPPER SKETCH formulation

We briefly describe the SKETCH formulation of the rule-synthesis problem in SWAPPER (Problem 3). In this section, we show how we can formulate constraints $1-3,5$ from Problem 3 in the SKETCH solver as a partial program. As discussed earlier in Subsec. 2.2.4, SKETCH will be used to solve the constraints $2,3,5$ from Problem 3 in SWAPPER because of the more efficient hybrid enumerative/SKETCH-based approach that incorporates the constraints $1,4$ as well.

We discuss the problem at hand using an example. Let us assume that (1) the pattern $or(lt(a,b),lt(a,d))$ is very common, so we want to find a rewrite rule for this pattern, and (2) this pattern often occurs in a context where a rewriter can prove that $b \leqslant 0$ and $d > 0$.

We encode semantics of the LHS in a straightforward manner using the operations used in the description of the pattern provided as an input:

```
#define NUM_INPUTS 3 //3 inputs a,b,d
#define LHS_SIZE 3 //or(lt(a,b),lt(a,d)) has 3 nodes
bit LHS(int a, int b, int d) {
    return (a < b || a < d);
}
```

We will reuse the macros `NUM_INPUTS` and `LHS_SIZE` again later to write the RHS grammar. We encode the predicate grammar as described earlier in Subsection 2.2.3. Each predicate is either a Boolean expression `boolExpr` or a disjunction of two such Boolean expressions. Each such Boolean expression is either `true` or based on an operation on two of the input variables (we omit the unary operator $\neg$ since there are no Boolean inputs). Note that a "`??`" inside an **if** is a hole that will be replaced with `true` or `false`, and any "`??`" inside a `generator` function can be replaced differently for each call of that function.

```
bit pred(int a, int b, int d) {
    if(??) boolExpr(a, b, d);
    else boolExpr(a, b, d) && boolExpr(a, b, d);
}

generator bit boolExpr(int a, int b, int d) {
    if(??) return true;
    else{
```

```
        int x = {| a | b | d |};
        int y = {| a | b | d |};
        return {| x ( < | > | == | != | <= | >= ) y |};
    }
}

void valid_under_assumption(int a, int b, int d) {
    //b<0 and d>0 assumption should imply that pred is true
    if (b<0 && d>0){
        assert(pred(a,b,d));
    }
}
```

Moreover, we assert the requirement for the predicate to be valid when the static assumption is true. Note that this requirement can be expanded to multiple static assumptions by requiring that the predicate `pred` is true, at least under one of the static assumptions. In general, this can be done in SKETCH by transforming constraint 1 from Problem 3 as follows. To solve

$$\bigvee_{j=1}^{m} \big(\forall x : assume_j(x) \implies pred(x)\big)$$

with SKETCH, we first convert the formula to its prenex normal form (requiring to move all quantifiers at the beginning):

$$\forall y_1 \forall y_2 \ldots \forall y_m \bigvee_{j=1}^{m} \big(assume_j(y_j) \implies pred(y_j)\big)$$

and then encode this new formula in SKETCH with m copies of each free variable in x. For example, suppose we have two contexts where the pattern $or(lt(a,b), lt(a,d))$ occurs: (1) $b \leqslant 0$ and $d > 0$ and (2) $a \leqslant 1$ and $d > 2$. The corresponding `valid_under_assumption` function in SKETCH will change to:

```
void valid_under_assumption(int a1, int b1, int d1,int a2, int b2, int d2) {
    bit assume1 = b1<=0 && d1>0;
    bit implication1 = !assume1 || pred(a1,b1,d1);
    bit assume2 = a2<=1 && d2>2;
    bit implication2 = !assume2 || pred(a2,b2,d2);
    assert(implication1 || implication2);
}
```

We encode the RHS template from Subsection 2.2.3 as follows: (1) the RHS pattern is formed by building an array (`vals`) of simple operations (`simpleOp`) (2) the i^{th} simple operation (`vals[i]`) is constructed by choosing an allowed operation and the operands of

that operation from the values `vals[0],vals[1],...,vals[i-1]`.

```
bit RHS(int a, int b, int d, ref int size) {
    int[LHS_SIZE+NUM_INPUTS-1] vals = 0;
    vals[0] = a;
    vals[1] = b;
    vals[2] = d;
    for (int i=NUM_INPUTS; i<LHS_SIZE+NUM_INPUTS-1; i++) {
        vals[i] = simpleOp(vals,i);
    }
    size = ??;
    return vals[size - NUM_INPUTS];
    //picks a value of size=?? such that this is a valid index in vals array
}

int simpleOp(int[LHS_SIZE+NUM_INPUTS] vals, int i){
    //use only the values from 0 to i - 1 indices as operands
    int choice_x = ??;
    int x = vals[choice_x];
    assert (choice_x >= 0 && choice_x < i);
    //to_bit converts an int to a bit
    if(??) return to_bit(x);
    if(??) return !to_bit(x);
    if(??) return -x;

    int choice_y = ??;
    int y = vals[choice_y];
    assert (choice_y >= 0 && choice_y < i);
    //second operand for binary operators
    if(??) {| x ( < | > | == | != | <= | >= ) y |};
    if(??) {| to_bit(x) ( && | || | ^ ) to_bit(y) |};
    //... and so on for other operators
    assert(false); //must choose one operator
}
```

Finally, we encode the correctness constraint for the rewrite rule as shown below:

```
void correctness(int a, int b, int d) {
    int rhs_size = 0;
    if (pred(a,b,d)) {
        assert(LHS(a,b,d) == RHS(a,b,d,rhs_size));
        assert(rhs_size < LHS_SIZE);
        minimize (rhs_size);
    }
}
```

Along with the correctness constraint, the function above also uses the **minimize** key-

word [105] from SKETCH's language to specify that the size of the RHS should be as small as possible (constraint 5 from Problem 3).

In this subsection, we showed how we can encode constraints $1-3, 5$ from Problem 3 in the SKETCH synthesizer to solve the rule-synthesis problem in SWAPPER. Now, we are ready to give the SKETCH implementation notes for EM-Synth.

A.3 SKETCH formulation for EM-GBF rule synthesis

The **EM-GBF** rule-synthesis SyGuS problem (Problem 4 and Subsec. 2.3.2) is translated to a partial program in SKETCH and fed to the SKETCH solver as an input. We showcase the partial programs used in SKETCH for this problem with an example. Note that we also use the custom solver inside SKETCH as described in Subsec. 2.3.3. This custom solver is specified in the partial programs using the name `customSynth`. The similarity function tables (Subsec. 2.3.3) are passed as a separate input (apart from the partial program) to SKETCH.

Example 5: Consider the two tables discussed in Example 1 (Sec. 1.2). The partial program that represents a Boolean formula matching rule (**GBF**) with $N_a = 7$ attribute-matching rules and $N_d = 3$ depth of grammar expansion is listed below.

```
//input bounds for the GBF grammar
#define N_a 7
#define N_d 3

generator bit attributeRule(int e){ // e = Example Id
    int i = ??; // Attribute Id
    assert (1 <= i && i <= 5);
    int f = ??; // Similarity function Id
    assert (1 <= f && f <= 29);
    //θ = customSynth(i,f);
    return (evalSimFn(e,i,f) >= customSynth(i,f));
}

generator bit gbfRule(int e, ref int A, int D){
    if (??){ A++; return attributeRule(e); }
    else {
        assert D >= 0;
        if (??) return ! (gbfRule(e,A,D − 1));
        else if (??) return gbfRule(e,A,D − 1) && gbfRule(e,A,D − 1);
        else return gbfRule(e,A,D − 1) || gbfRule(e,A,D − 1);
    }
```

```
}

bool matchingRule(int e){
  int A=0;
  int D=N_d;
  bool b = gbfRule(e,A,D);
  assert (A<=N_a);
  return b;
}

void examples(){
  //Example Id 1 is a positive example
  assert(matchingRule(1) == true);
  //Example Id 2 is a negative example
  assert(matchingRule(2) == false);
  ...
}
```

In the code above, some functions are annotated with being a `generator`. As described earlier, functions annotated with the `generator` keyword will be inlined with new holes (`??`) for every instantiation of this function. For example, `attributeRule` is a `generator` function, and every instantiation of this function will be free to pick different similarity functions. Since there are 5 aligned attributes, we assert that the values taken by i lie between 1 and 5. Similarly, the candidate space of 29 similarity functions is asserted accordingly. The values for threshold θ are chosen using a custom synthesis procedure. The function `evalSimFn` symbolically represents the evaluation of function f on attribute i of the records from example e (see more details in Subsec. 2.3.3). Also, `gbfRule` is a `generator` function with function `attributeRule` being inlined at most N_a times (enforced by a variable A passed by reference) and multiple recursive calls to itself to specify the possible expansion of the grammar. The expansion is bounded by a depth N_d passed as a parameter D to the generator `gbfRule`. Note again, in SKETCH, each `generator` function is completely inlined up to the specified depth as a parameter. This results in the *holes* ("`??`"s) occurring multiple times as well. Each hole inside the `if`'s represents a possible *true* or *false* value.

The `examples` function enforces the constraints using the `assert` keyword. The constraints assert that the resulting rule should work for the provided positive and negative examples. The SKETCH synthesizer will fill all the holes in the above partial program to synthesize a *complete program*, with a function `matchingRule` that represents a Boolean formula (**GBF**) for entity matching. □

Bibliography

[1] Amazon Web Services. Online. 98

[2] Boostpython library: quickly and easily export c++ to python. https://wiki.python.org/moin/boost.python. 110

[3] Datasketch: Minhash lsh. https://ekzhu.github.io/datasketch/lsh.html. 112

[4] Pyjnius: Python library for accessing java classes. http://pyjnius.readthedocs.io/en/latest/. 110

[5] Scikit learn: Support vector machines in practice. http://scikit-learn.org/stable/modules/svm.html. 112, 117

[6] Sketch: Working Manual. Online. 147

[7] Standardization, or mean removal and variance scaling. http://scikit-learn.org/stable/modules/preprocessing.html. 115

[8] SymPy: Python library for symbolic mathematics. Online. 86

[9] Tuning the hyper-parameters of an estimator. http://scikit-learn.org/stable/modules/grid_search.html. 112, 117

[10] 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. Syntax-guided synthesis. In *Dependable Software Systems Engineering*, pages 1–25. 2015. 23, 31, 33, 34, 36, 47, 98

[11] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. Practical and optimal lsh for angular distance. In *Proceedings of the 28th International Conference on Neural Information Processing Systems*, NIPS'15, pages 1225–1233, Cambridge, MA, USA, 2015. MIT Press. 108

[12] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman P. Amarasinghe. Opentuner: an extensible framework for program autotuning. In *PACT '14*. 81, 82, 83, 96

[13] Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In *CP '09*. 82

[14] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS XII, pages 394–403. ACM, 2006. 138

[15] H P Barendregt, M C J D Eekelen, J R W Glauert, J R Kennaway, M J Plasmeijer, and M R Sleep. Term graph rewriting. In *Volume II: Parallel Languages on PARLE: Parallel Architectures and Languages Europe*, pages 141–158, London, UK, UK, 1987. Springer-Verlag. 70

[16] Rohan Baxter, Peter Christen, and Tim Churches. A comparison of fast blocking methods for record linkage. In *SIGKDD Workshop on Data Cleaning, Record Linkage, and Object Consolidation*, pages 25–27, 2003. 139

[17] Omar Benjelloun, Hector Garcia-Molina, David Menestrina, Qi Su, Steven Euijong Whang, and Jennifer Widom. Swoosh: a generic approach to entity resolution. *VLDB J.*, 18(1), 2009. 139

[18] Philip A. Bernstein, Jayant Madhavan, and Erhard Rahm. Generic schema matching, ten years later. *PVLDB*, 4(11):695–701, 2011. 44

[19] Sahil Bhatia and Rishabh Singh. Automated correction for syntax errors in programming assignments using recurrent neural networks. *CoRR*, abs/1603.06129, 2016. 136

[20] Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Deep API programmer: Learning to program with apis. *CoRR*, abs/1704.04327, 2017. 136

[21] Mikhail Bilenko and Raymond J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *SIGKDD*, pages 39–48, 2003. 25, 139

[22] Roderick Bloem, Krishnendu Chatterjee, Karin Greimel, Thomas A. Henzinger, Georg Hofferek, Barbara Jobstmann, Bettina Könighofer, and Robert Könighofer. Synthesizing robust systems. *Acta Inf.*, 51(3-4):193–220, 2014. 136

[23] Roderick Bloem, Krishnendu Chatterjee, Thomas A. Henzinger, and Barbara Jobstmann. Better quality in synthesis through quantitative objectives. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 140–156. Springer, 2009. 136

[24] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008. 139

[25] Andrei Z. Broder. On the resemblance and containment of documents. In *In Compression and Complexity of Sequences (SEQUENCES 97*, pages 21–29. IEEE Computer Society, 1997. 108

[26] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. *Comput. Netw. ISDN Syst.*, 29(8-13):1157–1166, September 1997. 108

[27] Robert Brummayer and Armin Biere. Boolector: An efficient smt solver for bit-vectors and arrays. TACAS '09, pages 174–177, 2009. 21, 138

[28] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Conference OSDI '08*. 21, 138

[29] Luiz F. M. Carvalho, Alberto H. F. Laender, and Wagner Meira Jr. Entity matching: A case study in the medical domain. In *Proceedings of the 9th Alberto Mendelzon International Workshop on Foundations of Data Management*, 2015. 24

[30] Pavol Cerný, Krishnendu Chatterjee, Thomas A. Henzinger, Arjun Radhakrishna, and Rohit Singh. Quantitative synthesis for concurrent programs. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 243–259. Springer, 2011. 136

[31] Satish Chandra, Stephen J. Fink, and Manu Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *PLDI '09*, pages 363–374, 2009. 21, 138

[32] Krishnendu Chatterjee, Thomas A. Henzinger, Barbara Jobstmann, and Rohit Singh. Measuring and synthesizing systems in probabilistic environments. *J. ACM*, 62(1):9:1–9:34, 2015. 136

[33] Surajit Chaudhuri, Bee-Chung Chen, Venkatesh Ganti, and Raghav Kaushik. Example-driven design of efficient record matching queries. In *VLDB*, pages 327–338, 2007. 139

[34] Tianqi Chen. Introduction to boosted trees. *University of Washing Computer Science. University of Washington*, 22, 2014. 115

[35] Alvin Cheung, Owen Arden, Samuel Madden, Armando Solar-Lezama, and Andrew C. Myers. Statusquo: Making familiar abstractions perform using program analysis. In *CIDR*, 2013. 140

[36] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Partial replay of long-running applications. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 135–145. ACM, 2011. 21, 138

[37] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Inferring sql queries using program synthesis. *CoRR*, abs/1208.2013, 2012. 23

[38] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Using program synthesis for social recommendations. In *CIKM*, pages 1732–1736, 2012. 140

[39] William W. Cohen and Jacob Richman. Learning to match and cluster large high-dimensional data sets for data integration. In *SIGKDD*, pages 475–480, 2002. 139

[40] Nilesh Dalvi, Vibhor Rastogi, Anirban Dasgupta, Anish Das Sarma, and Tamás Sarlós. Optimal hashing schemes for entity matching. In *WWW*, pages 295–306, 2013. 139

[41] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. TACAS'08/ETAPS'08, pages 337–340. Springer-Verlag, 2008. 12, 21, 73, 74, 75, 89, 132

[42] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. Program synthesis using natural language. May 2016. 33, 137

[43] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel rahman Mohamed, and Pushmeet Kohli. RobustFill: Neural program learning under noisy I/O. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 990–998, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. 136

[44] Hong Hai Do and Erhard Rahm. COMA - A system for flexible combination of schema matching approaches. In *VLDB*, 2002. 25

[45] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. Duplicate record detection: A survey. *IEEE Trans. Knowl. Data Eng.*, 19(1):1–16, 2007. 25, 107

[46] Kave Eshghi and Shyamsundar Rajaram. Locality sensitive hash functions based on concomitant rank order statistics. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 221–229, New York, NY, USA, 2008. ACM. 108

[47] Ivan Fellegi and Alan Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64 (328), 1969. 25, 139

[48] Donatella Firmani, Barna Saha, and Divesh Srivastava. Online entity resolution using an oracle. *PVLDB*, 9(5):384–395, 2016. 140

[49] Martin A. Fischler and Robert C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, June 1981. 31, 63, 65

[50] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Dpll (t): Fast decision procedures. In *CAV*, volume 4, pages 175–188. Springer, 2004. 141

[51] Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack, and Adam M. Szalkowski. GrGen: A Fast SPO-Based Graph Rewriting Tool. pages 383 – 397, 2006. 139

[52] Nithin George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J. Brown, Arvind K. Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. Hardware system synthesis from domain-specific languages. In *24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany, 2-4 September, 2014*, pages 1–8, 2014. 33, 137

[53] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. 2017. 136

[54] Chaitanya Gokhale, Sanjib Das, AnHai Doan, Jeffrey F. Naughton, Narasimhan Rampalli, Jude Shavlik, and Xiaojin Zhu. Corleone: Hands-off crowdsourcing for entity matching. In *SIGMOD*, 2014. 112, 116, 139, 140

[55] Bernhard Gramlich. *Termination and confluence properties of structured rewrite systems*. Universität Kaiserslautern. Fachbereich Informatik, 1996. 70

[56] Sumit Gulwani. Programming by examples. *Dependable Software Systems Engineering*, 45:137, 2016. 3, 15, 16, 20, 135

[57] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. volume 55, pages 97–105, January 2012. 33, 137

[58] R. Nigel Horspool. Incremental generation of lr parsers. *Computer languages*, 15:205–233, 1989. 139

[59] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. Paramils: An automatic algorithm configuration framework. *J. Artif. Int. Res.*, 2009. 82

[60] Frank Hutter, Marius Thomas Lindauer, Adrian Balint, Sam Bayless, Holger H. Hoos, and Kevin Leyton-Brown. The configurable SAT solver challenge (CSSC). *CoRR*, abs/1505.01221, 2015. 82

[61] Jeevana Priya Inala, Xiaokang Qiu, Ben Lerner, and Armando Solar-Lezama. Type assisted synthesis of recursive transformers on algebraic data types. *CoRR*, abs/1507.05527, 2015. 23

[62] Jeevana Priya Inala, Rohit Singh, and Armando Solar-Lezama. Synthesis of domain specific CNF encoders for bit-vector solvers. In *SAT 2016*, 2016. 16, 20, 23, 29, 98, 102

[63] Jinseong Jeon, Xiaokang Qiu, Jeffrey S. Foster, and Armando Solar-Lezama. Jsketch: sketching for java. In Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. ACM, 2015. 23

[64] Jianqiu Ji, Jianmin Li, Shuicheng Yan, Qi Tian, and Bo Zhang. Min-max hash for jaccard similarity. In *Data Mining (ICDM), 2013 IEEE 13th International Conference on*, pages 301–309. IEEE, 2013. 108

[65] Neil C. Jones and Pavel A. Pevzner. *An Introduction to Bioinformatics Algorithms (CMB)*. MIT Press, August 2004. 55

[66] Lars Kolb, Andreas Thor, and Erhard Rahm. Dedoop: Efficient deduplication with hadoop. *PVLDB*, 5(12):1878–1881, 2012. 139

[67] Mark A. Kon, Yue Fan, Dustin Holloway, and Charles DeLisi. Svmotif: A machine learning motif algorithm. In *Proceedings of the Sixth International Conference on Machine Learning and Applications*, ICMLA '07, pages 573–580. 138

[68] Hanna Köpcke and Erhard Rahm. Training selection for tuning entity matching. In *International Workshop on Quality in Databases and Management of Uncertain Data*, pages 3–12, 2008. 110

[69] Hanna Köpcke, Andreas Thor, and Erhard Rahm. Evaluation of entity resolution approaches on real-world match problems. *PVLDB*, 3(1):484–493, 2010. 110, 114

[70] Himabindu Lakkaraju, Stephen H. Bach, and Jure Leskovec. Interpretable decision sets: A joint framework for description and prediction. In *KDD*, 2016. 25

[71] Tao Lei, Regina Barzilay, and Tommi S. Jaakkola. Rationalizing neural predictions. In *EMNLP*, 2016. 25

[72] Alberto Leon-Garcia. *Probability, Statistics, and Random Processes for Electrical Engineering*. Pearson/Prentice Hall, third edition, 2008. 58

[73] Nikolaj Bjørner Leonardo de Moura. Smt: Techniques, hurdles, applications. *SAT/SMT Summer School, MIT*, 2011. 138

[74] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge university press, 2014. 112

[75] Yongchao Liu, Bertil Schmidt, and Douglas L. Maskell. An ultrafast scalable many-core motif discovery algorithm for multiple gpus. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11. 138

[76] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with alive. *SIGPLAN Not.*, 50(6):22–32, June 2015. 137

[77] Andrew McCallum, Kamal Nigam, and Lyle H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *SIGKDD*, pages 169–178, 2000. 139

[78] Andrew Kachites McCallum, Kamal Nigam, Jason Rennie, and Kristie Seymore. Automating the construction of internet portals with machine learning. *Information Retrieval*, 3(2):127–163, 2000. 114

[79] E. J. Mccluskey. Minimization of Boolean functions. *The Bell System Technical Journal*, 35(5):1417–1444, November 1956. 78, 86

[80] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. Dag-aware aig rewriting a fresh look at combinational logic synthesis. In *Proceedings of the 43rd annual Design Automation Conference*. ACM, 2006. 95, 97

[81] Barzan Mozafari, Purnamrita Sarkar, Michael J. Franklin, Michael I. Jordan, and Samuel Madden. Scaling up crowd-sourcing to very large datasets: A case for active learning. *PVLDB*, 8(2):125–136, 2014. 140

[82] David R. Musicant, Vipin Kumar, and Aysel Ozgur. Optimizing f-measure with support vector machines. In *Proc. of the 16th International Florida Artificial Intelligence Research Society Conference*, pages 356–360, 2003. 117

[83] Andriy Nikolov, Mathieu d'Aquin, and Enrico Motta. Unsupervised learning of link discovery configuration. In *ESWC*, pages 119–133, 2012. 140

[84] Emilio Parisotto, Abdelrahman Mohamed, Rishabh Singh, Lihong Li, Denny Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. In *ICLR 2017*, February 2017. 136

[85] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. 112, 117

[86] Mikael Pettersson. A term pattern-match compiler inspired by finite automata theory. In *CC' 92*, pages 258–270. Springer-Verlag, 1992. 76

[87] Phitchaya Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. Greenthumb: Superoptimizer construction framework. Technical Report UCB/EECS-2016-8, EECS Department, University of California, Berkeley, Feb 2016. 138

[88] Nadia Polikarpova, Jean Yang, Shachar Itzhaky, and Armando Solar-Lezama. Type-driven repair for information flow security. *CoRR*, abs/1607.03445, 2016. 33, 137

[89] Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. sk_p: a neural program corrector for moocs. *CoRR*, abs/1607.02902, 2016. 136

[90] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. Learning programs from noisy data. In *POPL*, pages 761–774, 2016. 3, 15, 16, 20, 135

[91] Jakob Rehof and Moshe Y. Vardi. Design and Synthesis from Components (Dagstuhl Seminar 14232). *Dagstuhl Reports*, 4(6):29–47, 2014. 16, 17

[92] Geir Kjetil K. Sandve and Finn Drabløs. A survey of motif discovery methods in an integrated framework. *Biology direct*, April 2006. 55, 138

[93] Sunita Sarawagi and Anuradha Bhamidipaty. Interactive deduplication using active learning. In *KDD*, pages 269–278, 2002. 139, 140

[94] Len Seligman, Peter Mork, Alon Y. Halevy, Kenneth P. Smith, Michael J. Carey, Kuang Chen, Chris Wolf, Jayant Madhavan, Akshay Kannan, and Doug Burdick. Openii: an open source information integration toolkit. In *SIGMOD*, 2010. 25

[95] Rishabh Singh. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *PVLDB*, 9(10):816–827, 2016. 16, 135

[96] Rishabh Singh. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *PVLDB*, 9(10):816–827, 2016. 140

[97] Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *CAV*, pages 634–651, 2012. 30

[98] Rishabh Singh and Sumit Gulwani. Predicting a correct program in programming by example. In *Computer-Aided Verification (CAV)*, pages 398–414, 2015. 135

[99] Rishabh Singh and Sumit Gulwani. Transforming spreadsheet data types using examples. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,*

POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016, pages 343–356. ACM, 2016. 16, 135

[100] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. *SIGPLAN Not.*, 48(6):15–26, June 2013. 23, 98

[101] Rishabh Singh and Pushmeet Kohli. AP: artificial programming. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*, volume 71 of *LIPIcs*, pages 16:1–16:12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. 136

[102] Rishabh Singh and Armando Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *SIGSOFT FSE*, 2011. 23

[103] Rohit Singh. Synthesizing a synthesis tool. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2013. 76

[104] Rohit Singh, Vamsi Meduri, Ahmed K. Elmagarmid, Samuel Madden, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Armando Solar-Lezama, and Nan Tang. Generating concise entity matching rules. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1635–1638. ACM, 2017. 16, 20, 29

[105] Rohit Singh, Rishabh Singh, Zhilei Xu, Rebecca Krosnick, and Armando Solar-Lezama. Modular synthesis of sketches using models. In *VMCAI 2014*. 43, 150

[106] Rohit Singh and Armando Solar-Lezama. Swapper: A framework for automatic generation of formula simplifiers based on conditional rewrite rules. In *2016 Formal Methods in Computer-Aided Design (FMCAD)*, pages 185–192, Oct 2016. 16, 20, 29

[107] Parag Singla and Pedro Domingos. Entity resolution with markov logic. In *ICDM*, pages 572–582, 2006. 139

[108] Anirudh Sivaraman, Mihai Budiu, Alvin Cheung, Changhoon Kim, Steve Licking, George Varghese, Hari Balakrishnan, Mohammad Alizadeh, and Nick McKeown. Packet transactions: A programming model for data-plane algorithms at hardware speed. *CoRR*, abs/1512.05023, 2015. 23

[109] Tony C. Smith and Eibe Frank. *Statistical Genomics: Methods and Protocols*, chapter Introducing Machine Learning Concepts with WEKA, pages 353–378. Springer, New York, NY, 2016. 112

[110] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, Berkeley, CA, USA, 2008. AAI3353225. 43

[111] Armando Solar-Lezama. The sketching approach to program synthesis. In *APLAS*, pages 4–13, 2009. 23, 33, 36, 40, 137

[112] Armando Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013. 30, 63

[113] Mária Svoreňová and Marta Kwiatkowska. Quantitative verification and strategy synthesis for stochastic games. *European Journal of Control*, 30:15–30, 2016. 137

[114] Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. Transit: specifying protocols with concolic snippets. *ACM SIGPLAN Notices*, 48(6):287–296, 2013. 33, 137

[115] Jiannan Wang, Tim Kraska, Michael J. Franklin, and Jianhua Feng. Crowder: Crowdsourcing entity resolution. *PVLDB*, 5(11):1483–1494, 2012. 140

[116] Jiannan Wang, Guoliang Li, Jeffrey Xu Yu, and Jianhua Feng. Entity matching: How similar is similar. *PVLDB*, 4(10), 2011. 3, 48, 110, 114

[117] Jiannan Wang, Guoliang Li, Jeffrey Xu Yu, and Jianhua Feng. Entity matching: How similar is similar. *PVLDB*, 4(10), 2011. 31, 46, 110, 112, 115, 116, 125, 139, 140

[118] Jingdong Wang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji. Hashing for similarity search: A survey. *CoRR*, abs/1408.2927, 2014. 107, 108, 130, 132

[119] Jean Yang. *Preventing information leaks with policy-agnostic programming.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2015. 33, 137

[120] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, October 2016. 89