Program Synthesis over Noisy Data

by

Shivam Handa

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

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2022

Author .....
Department of Electrical Engineering and Computer Science
July 29, 2022

Certified by .....
Martin Rinard
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

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

Program Synthesis over Noisy Data

by

Shivam Handa

Submitted to the Department of Electrical Engineering and Computer Science
on July 29, 2022, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

I present a new framework and associated synthesis algorithms for program synthesis over noisy data, i.e., data that may contain incorrect/corrupted input-output examples. I model the process that produced the noisy dataset as the selection of inputs and a hidden program from an input source and program source followed by the application of a noise source to the correct outputs from the hidden program to obtain the noisy dataset. This model makes it possible to formulate the problem of noisy program synthesis as an optimization problem formulated over the loss of a candidate program over the noisy dataset and the complexity of the candidate program.

I present a noisy program synthesis algorithm based on finite tree automaton. Results from an implemented system running this algorithm on problems from the SyGuS 2018 benchmark suite highlight the algorithm's ability to successfully synthesize programs in the face of noisy data.

I extend the noisy program synthesis framework to formally define the concepts of an optimal loss function and the convergence of a program synthesis algorithm to a correct program. Working with these concepts, I present optimal loss functions and convergence results for a wide range of program synthesis problems in the text manipulation domain, including results that characterize optimality and convergence properties of noise sources and loss functions used in experiments with the implemented synthesis algorithm. These results provide insight into the reasons for the success of the presented technique and can help enable the development of effective loss functions and noisy program synthesis algorithms in a range of contexts.

I also present a new noisy program synthesis algorithm that uses an abstraction refinement based optimization process to synthesize programs. The presented experimental results demonstrate the significant performance improvements that this new technique can deliver. Building on this abstraction refinement technique, I present new noisy program synthesis algorithms that can work with both noisy inputs and noisy outputs as well as domain specific languages that include infinite sets of constants.

Thesis Supervisor: Martin Rinard
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to give my thanks to my supervisor Prof. Martin Rinard who made this work possible. His guidance and advice carried my through all stages of my Ph.D study and research. His stories for me was what sustained and motivated me throughout this process.

Besides my advisor, I would like to thank the rest of my thesis commitee: Prof. Michael Carbin and Prof. Armando Solar-Lezema for their brilliant comments and suggestions.

Last but not the least, I would also like to give a special thanks to my parents Sanjay Handa and Geeta Handa, for supporting me throughout my life.

This doctoral thesis has been examined by a Committee of the Department of Electrical Engineering and Computer Science as follows:

Professor Armando Solar-Lezema .....
Chairman, Thesis Committee
Professor of Electrical Engineering and Computer Science

Professor Martin Rinard .....
Thesis Supervisor
Professor of Electrical Engineering and Computer Science

Professor Michael Carbin .....
Member, Thesis Committee
Assistant Professor of Electrical Engineering and Computer Science

Contents

Chapter 1

Introduction

In recent years, there has been significant interest in learning programs from input-output examples. These techniques have been successfully used to synthesize programs for domains such as string and format transformations [20, 35], data wrangling [15], data completion [43], and data structure manipulation [16, 27, 47]. Even though these efforts have been largely successful, they generally do not aspire to work with noisy datasets that may contain corrupted input-output examples.

In the real world, data contains noise [45, 34, 19, 21]. To extract information from a noisy dataset, users either 1) clean the dataset to remove noise and use a technique that works on noise-free data to extract information [19, 45, 34], or 2) use a technique which can tolerate noise and can extract information from the noisy dataset [21]. Using these methods, the user aims to learn information the user would have learned over noise-free data.

Given a dataset and a domain specific language (DSL), program synthesis techniques generally aim to synthesize a program which 1) is accepted by the DSL and 2) satisfies all input-output examples (i.e., given an input in the dataset, the program produces the corresponding output in the dataset) [16, 27, 47, 43, 42]. When examples contain noise, these techniques typically find it impossible to synthesize a program which satisfies all of the input-output examples. Even if the synthesis algorithm is able to synthesize a program which satisfies the noisy input-output examples, this synthesized program may not be the desired or most likely program given the noisy

dataset.

I present a framework and associated program synthesis algorithms that, given a potentially noisy dataset of input-output examples, are designed to synthesize a program which *best-fits* the noisy dataset. I start by formalizing a model of the process which generated the noisy dataset (Chapter 2). In this model a program source randomly selects a hidden program from a given DSL. An input source randomly generates n inputs. The hidden noise free outputs are the outputs generated by the hidden program over these inputs. A noise source then corrupts these noise free outputs to construct corresponding noisy outputs. Only the original inputs and the noisy outputs are visible to the synthesis algorithm.

I formalize the concept of *best-fit* using the concepts of a loss function, a regularizer, a complexity measure, and an objective function. Given a dataset and a program, the loss function measures the cost of the input-output examples on which the program produces a different output than the output in the dataset. Given inputs and a program, a regularizer penalizes the program based on the probability that a random program in our search space produces the same output as the given program on the given inputs. This allows us to bias the synthesis algorithm away from programs which produce spurious or erroneous outputs. Given a program, the complexity measure measures how complex a program is. This measure makes it possible to bias the synthesis algorithm towards simpler programs. The objective function combines all these scores to return a combined score. The objective function allows us to trade off between these scores. For example, similar to machine learning, we can trade off between the loss of a program and the complexity of a program, i.e., stop a program from overfitting the data by synthesizing a simpler program. This turns the problem of synthesizing the *best-fit* program into a problem of synthesizing a program which solves an optimization problem, i.e., synthesizing the program which minimizes the objective function.

1.1 Tree Automaton Based Synthesis Technique

I present a new synthesis algorithm which uses finite tree automata (FTA) to synthesize the *best fit* program (Chapter 3). Given a dataset $\mathcal{D}$, the synthesis algorithm uses a finite tree automaton to partition programs and sub-expressions, as defined by a grammar, into equivalence classes. Each accepting state in the finite tree automaton corresponds to a set of programs, in the given grammar, which map inputs in dataset $\mathcal{D}$ to the same output. For all programs in the given grammar, there exists exactly one accepting state which accepts this program. All programs accepted by a given accepting state have the same input-output behavior over dataset $\mathcal{D}$. Therefore, they have the same loss and regularizer weight over the dataset $\mathcal{D}$. My technique uses dynamic programming to find the minimum complexity program accepted by a given state. For a given accepting state, the simplest program has the minimum objective function value over all programs accepted by this state. Since all programs in the given grammar are accepted by exactly one accepting state, the technique iterates over all accepting states and uses the simplest program accepted by each state to synthesize the program which minimizes the objective function.

1.2 Experimental Results

I have implemented this technique and applied it to various synthesis problems in the SyGuS 2018 benchmark set [1]. I present the results of these experiments in Chapter 4. The results indicate that the technique is effective at solving program synthesis problems over strings with modestly sized solutions even in the presence of substantial noise. For discrete noise sources and a loss function that is a good match for the noise source, the technique is typically able to extract enough information left intact in corrupted outputs to synthesize a correct program even when all outputs are corrupted (I consider a synthesized program to be correct if it agrees with all input-output examples in the original hidden noise-free dataset). Overall the results highlight the potential for effective program synthesis even in the presence of substantial noise.

1.3 Optimal Noisy Program Synthesis

My experiments show that the choice of the loss function can have a significant impact on the effectiveness of the synthesis algorithm. This phenomenon motivated me to formalize the conditions under which a loss function makes it possible to synthesize the correct program with high probability. The effectiveness of the loss function depends on the noise source which corrupted the dataset.

Chapter 5 presents a Bayesian inference based framework to characterize the effectiveness of noisy program synthesis under a variety of conditions. Together, the components of this framework provide guidance for selecting a good loss function given information about the context in which the loss function will be deployed. It also identifies situations in which noisy program synthesis is not possible, for example because the noise source that corrupts the input/output examples destroys too much information for *any* synthesis algorithm to succeed.

1.4 Convergence

Successful noisy program synthesis may be impossible if the combination of noise source and loss function fails to preserve enough information to differentiate programs with different behaviors. I approach this issue by considering *convergence*, specifically the conditions under which a noisy program synthesis algorithm can identify a program p with identical behavior as the hidden program p_h given enough input/output examples.

I model the set of available input/output examples via an *input source*, which is a probability distribution $\rho_i(\boldsymbol{x})$ over inputs $\boldsymbol{x}$ that models the probability of selecting inputs $\boldsymbol{x}$ from the set of available inputs. The first requirement for successful convergence is that the input source must be *differentiating*, i.e., given two programs p and p_h with different behaviors on some inputs, the probability of distinguishing the programs given enough inputs approaches one. More formally, for all $\delta > 0$ and $\epsilon > 0$, there must exist a dataset size k such that for all $n > k$, sampling inputs $\boldsymbol{x}$ of length n from $\rho_i(\boldsymbol{x}|n)$ and computing the distance between the corresponding outputs $p[\boldsymbol{x}]$ and $p_h[\boldsymbol{x}]$ must produce a distance greater than ϵ with at least probability $1 - \delta$.

By itself, a differentiating noise source does not ensure convergence. We must also have a noise source and corresponding loss function pair that together preserve the ability of the input source to distinguish programs with different behaviors. I therefore also characterize the conditions on the noise source and loss function that ensure that they can distinguish programs with different behaviors. A key convergence theorem (Theorem 7) establishes that any synthesis algorithm that minimizes the loss function and works with a differentiating input source and a differentiating noise source/loss function pair converges.

1.5 Abstraction Refinement in Noisy Program Synthesis

I present a new noisy program synthesis algorithm based on (a generalization of) abstraction refinement in Chapter 6. This algorithm also uses a finite tree automaton to partition the space of the programs. Each accepting state in the finite tree automaton is associated with a set of output vectors. This set is expressed using abstract values. Each accepting state corresponds to a set of programs. These programs map inputs in $\mathcal{D}$ to outputs within the set associated with this state. A program, in our search space, corresponds to exactly one accepting state. An abstract value allows us to compute the minimum loss value and regularizer weight over all outputs represented by this abstract value. Given an accepting state, the technique computes a lower bound of the objective function value over all programs accepted by this state. By iterating over all accepting states, the technique selects the state with the minimum lower bound value, then synthesizes the simplest program accepted by this state. This program is the candidate program.

If the objective function value of the candidate program is equal to the lower bound value, then the candidate program minimizes the objective function. The technique therefore halts and synthesizes this candidate program.

If the objective function value of the candidate program is greater than the lower bound, then this candidate program may not minimize the objective function. In this case there exists at least one input-output example such that the candidate program's loss/regularizer weight on this counterexample is greater than the lower

bound loss/regularizer weight on this counterexample. Based on this counterexample, the technique then refines the abstract values in the finite tree automaton, further partitioning the space of programs and improving the lower bound on the objective function value for each partition. My technique then repeats this process until it halts. My technique is **guaranteed** to halt.

I have implemented my algorithm in the *Rose* synthesis tool. *Rose* can be instantiated to work in different domains by providing suitable domain specific languages, abstract semantics, and concrete semantics of functions within the language. *Rose* is parameterized over a large class of objective functions, loss functions, and complexity measures. I benchmark *Rose* using the SyGuS 2018 benchmark suite [3]. The empirical evaluation demonstrates that *Rose* is significantly faster than my tree automaton based technique.

1.6 Domain Specific Languages With Infinite Sets of Constants

The noisy program synthesis framework presented in Chapters 3 and 6 only work on program spaces which are finite, including programs spaces which can only draw upon constants from a small finite set of constants. This is a standard practice in enumeration/non-solver based program synthesis techniques [43, 42, 3]. In Chapter 7, I extend the framework to deal with program spaces which can draw constants from a large (potentially infinite) set.

I modify my abstraction refinement based synthesis algorithm that synthesizes programs drawn from domain specific languages with infinite sets of constants. The algorithm partitions the space of constants using a set of predicates, treating each partition as an abstract value. The algorithm then uses these partitions to build a finite tree automaton. Each accepting state is associated with a set of programs (where programs are built using a set of constants represented using abstract values) which map the inputs in the dataset to the same set of abstract value outputs.

On a higher level, the rest of my algorithm proceeds in a similar manner to my abstraction refinement based synthesis algorithm. I prove that the algorithm is guar-

anteed to halt. When the algorithm halts, the algorithm synthesizes the program which minimizes the objective function.

I have implemented the algorithm in the *Rose* synthesis tool and instantiated *Rose* to work with a DSL that contains programs that manipulate text. This DSL allows its programs to use arbitrary strings as constants.

1.7 Datasets With Noisy Inputs

In Chapter 8, I extend the framework to deal with both noisy inputs and noisy outputs. I also present a modified version of my abstraction refinement based synthesis algorithm to synthesize programs over noisy inputs and noisy outputs.

Similar to my approach in dealing with noisy outputs, I formalize this problem as an optimization problem. The original algorithm for noisy outputs but noise-free inputs computes the output loss and the regularizer weight. To compute these measures, the algorithm also computes the noise-free outputs of candidate programs in our search space. With noisy inputs, a challenge is that the algorithm does not have access to the noise-free inputs to compute the noise-free outputs. I overcome this challenge by parameterizing the optimization problem with a program and also a set of noise-free inputs. The goal of the synthesis algorithm, in this case, is to synthesize a program and a set of noise-free inputs which *best-fit* the noisy dataset.

I introduce the concept of an input loss function which measures the distance between the noisy inputs in the dataset and our predicted noise-free inputs. A modified objective function makes it possible to trade off the input loss, the output loss, the regularizer, and the complexity. The ability to trade off the input loss and output loss is important, as it prevents the synthesis algorithm from overfitting the noisy inputs. It also prevents the synthesis algorithm from synthesizing a program which overfits the noisy outputs, by selecting arbitrary noise-free inputs.

I modify my abstraction refinement based synthesis algorithm to synthesize the program which minimizes such objective functions. The algorithm partitions the space of all possible inputs using a set of predicates. These partitions serve as abstract input values for partitioning the space of programs. The algorithm constructs these

partitions by building a finite tree automaton on these abstract input values. Each accepting state in the finite tree automaton is associated with a set of abstract output values. A program is accepted by an accepting state if the program maps the abstract input values to these abstract output values. For an accepting state and a mapping between noisy input-output examples to input partitions, we can estimate a lower bound on the objective function value for each program in a given partition based on this mapping. For an accepting state, the algorithm synthesizes a mapping, such that, the mapping minimizes the lower bound across all possible mappings. The algorithm iterates through all accepting states and selects an accepting state and associated mapping which minimizes the lower bound. The algorithm computes the simplest program accepted by this accepting state as the candidate program. Using this mapping, the algorithm computes the candidate noise-free input for each noisy input. For a noisy input, the candidate noise-free input is the input drawn from the noisy input's mapped partition that minimizes the input loss.

If the objective function value of the candidate program and noise-free inputs is equal to the lower bound of the accepting state's objective function value then our technique halts and synthesizes this program. This program minimizes the objective function.

Otherwise, the algorithm generates a counterexample to refine the input partitions and the finite tree automaton. The candidate program's loss/regularizer weight on this counterexample is greater than the lower bound loss/regularizer weight (over all programs accepted by the candidate accepting state) on this counterexample. My algorithm then refines the input partitions and the finite tree automaton to further improve our estimates of the lower bound. My algorithm then repeats this process until it halts. I prove that the algorithm is guaranteed to halt. When the algorithm halts, it synthesizes a program which minimizes the objective function.

1.8 Contributions

This thesis makes the following contributions:

- **Noisy Program Synthesis and Correctness:** It presents and formalizes a novel framework for noisy program synthesis. The framework introduces the concepts of a program source, an input source, and a noise source to formalize the program synthesis problem over noisy data. This formalization uses the concepts of a loss function, regularizer, complexity measure, and objective function to define correct noisy program synthesis as an optimization problem over the objective function.

- **Tree Automaton based Synthesis Algorithm:** It presents a novel noisy program synthesis algorithm. This algorithm uses finite tree automaton to synthesize programs that minimize the objective function.

- **Experimental Results:** It presents experimental results from the novel tree automaton based noisy program synthesis algorithm on the SyGuS 2018 benchmark set. These results characterize the scalability of the technique and highlight interactions between the DSL, the noise source, the loss function, and the overall effectiveness of the synthesis technique. In particular, they highlight the ability of the technique to often (given a close match between the noise source and the loss function) synthesize a correct program p even when 1) there are only a handful of input-output examples in the data set $\mathcal{D}$ and 2) all outputs are corrupted.

- **Optimal Loss Functions:** The noisy program synthesis algorithm uses loss functions to drive the search process for the *best-fit* program. The thesis demonstrates how to design optimal loss functions when: 1) we are given a noise source in the form of conditional probability distribution that, given correct outputs, characterizes the probability of obtaining corresponding noisy outputs, 2) we are given only a prior probability distribution over possible noise sources, or 3) the noise source is any one of a number of noise sources appropriate for text manipulation problems.

- **Convergence:** Convergence is an important property which is well studied in the statistics and machine learning literature [25, 17]. Given a synthesis setting (i.e., a set of programs, an input source, a noise source, a program source, and a loss function), the thesis defines a convergence property that makes it possible to guarantee that, given a large enough random dataset, the synthesis algorithm will synthesize a program equivalent to a conceptual hidden correct program with high probability. It identifies conditions that ensure convergence, including results that guarantee convergence in a range of noisy program synthesis contexts.

- **Text Manipulation:** The thesis presents general noise sources, corresponding optimal loss functions, and convergence results appropriate for a wide range of text manipulation contexts. These results provide insight into the empirical results presented in this thesis.

- **Abstraction Refinement Technique:** It presents a new program synthesis technique for synthesizing programs over noisy datasets. This technique uses the abstract semantics of DSL constructs to partition the program search space. For each partition, the technique uses the abstract semantics to compute an abstract value representing the outputs of all programs in that partition. The abstract value also allows the technique to soundly estimate the minimum possible loss value over all programs in each partition. It presents a new refinement technique that works with the sound approximation of the minimum loss values to refine the current partition, then discard partitions that cannot possibly contain the optimal program. Iteratively applying this refinement technique delivers a program which optimizes the objective function. It also presents the *Rose* synthesis system, which implements the abstraction refinement based algorithm for the string domain. The experimental evaluation shows that *Rose* delivers substantial speedups over our original tree automaton based technique.

- **Synthesis over Domain Specific Languages with an Infinite Set of Constants:** It formalizes a new framework for noisy program synthesis over

languages with infinite sets of constants. These languages have programs with constant sub-expressions drawn from this infinite set of constants. It presents a modification to our abstraction refinement based synthesis algorithm that can synthesize programs over such languages. It also evaluates this algorithm on text manipulation languages with string constants.

- **Synthesis over Noisy Inputs:** It formalizes a new framework for noisy program synthesis over datasets containing both noisy outputs and noisy inputs. It introduces a concept of input loss function and formalizes the synthesis of the *best-fit* program as an optimization problem. It presents a modification to our abstraction refinement based synthesis algorithm that can synthesize programs even in the presence of input noise. The presented experimental results show that the algorithm can synthesize the correct program over datasets containing input and output noise.

Data in the real world contains noise. This thesis presents a novel framework and associated algorithms to synthesize programs over datasets containing noise. I believe the framework, algorithms, and results presented in this thesis will pave the way to formulate and design robust methods which can synthesize programs over real world data.

Chapter 2

Noisy Program Synthesis

I first introduce the concept of a noisy dataset and the problem of program synthesis over noisy datasets. I formalize the concept of a noisy dataset using the concept of a program source, which generates a hidden underlying program, an input source, which generates a set of n inputs, hidden outputs, which are computed by the hidden program using the generated inputs, and a noise source, which corrupts these hidden outputs to produce a set of noisy outputs. Using these concepts, I formalize the concept of synthesizing the *best-fit* program over the noisy dataset as an optimization problem.

2.1 Noisy Dataset and Program Synthesis

Let $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$ be a dataset, where $\boldsymbol{x} = \langle x_1, \ldots x_n \rangle$ are the inputs and $\boldsymbol{y} = \langle y_1, \ldots y_n \rangle$ are the noisy outputs. The **program space** G is a **set of programs** containing the **hidden underlying program** p_h (i.e., $p_h \in G$), which generated the noise-free dataset, which was then corrupted by a noise source to create the noisy dataset. I use the notation ρ_p to denote a **program source**, which is a prior probability distribution over programs in G. The hidden program p_h is sampled from the prior distribution ρ_p.

Input Source: An input source is a probabilistic process which generates the inputs provided to the hidden underlying program. Formally, an input source is a probability distribution ρ_i, from which n inputs $\boldsymbol{x} = \langle x_1, \ldots x_n \rangle$ are sampled with probability

$\rho_i(\boldsymbol{x} \mid n)$.

Given a program $p \in G$ and an input x, I use the notation $p(x)$ to denote the output of program p when executed on input x. Given a vector of input values $\boldsymbol{x} = \langle x_1, x_2, \ldots x_n \rangle$, I use the notation $p[\boldsymbol{x}]$ to denote vector $\langle p(x_1), p(x_2), \ldots p(x_n) \rangle$.

Noise Source: A noise source is a probabilistic process which corrupts the correct outputs returned by the hidden program to create the noisy outputs. Formally, a noise source ρ_N is a probability distribution. Given a hidden program p_h and outputs $\boldsymbol{z} = \langle z_1, \ldots z_n \rangle$, the noisy outputs $\boldsymbol{y} = \langle y_1, \ldots y_n \rangle$ are sampled from the probability distribution ρ_N, with probability $\rho_N(\boldsymbol{y} \mid \boldsymbol{z})$.

Equivalent Programs: Given two programs $p_1, p_2 \in G$, I use the notion $p_1 \approx p_2$ to denote program p_1 is equivalent to p_2, i.e., for all $x \in X$, $p_1(x) = p_2(x)$.

Noisy Dataset: A noisy dataset $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$ is composed of a set of input values, denoted by $\boldsymbol{x}$, and corresponding noisy output values, denoted by $\boldsymbol{y}$. I assume the dataset $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$ of size n is constructed by the following process:

- A hidden program $p_h \in G$ is sampled from the probability distribution ρ_p.
- n inputs $\boldsymbol{x} = \langle x_1, \ldots x_n \rangle$ are sampled from probability distribution $\rho_i(\cdot \mid n)$.
- The process compute outputs $\boldsymbol{z} = \langle z_1, \ldots z_n \rangle$, where $z_i = p_h(x_i)$.
- The noise source introduces noise by corrupting outputs $\boldsymbol{z}$ to $\boldsymbol{y}$ with probability $\rho_N(\boldsymbol{y} \mid \boldsymbol{z})$

A potential goal, that one can explore, is the problem of synthesizing the hidden program p_h, or any program equivalent to p_h. Because of the probabilistic nature of the noise process, given inputs $\boldsymbol{x}$, noisy outputs $\boldsymbol{y}$, program space G, and nothing else, it may be impossible to infer any information about the hidden process. Even if we are given the program source ρ_p, the input source ρ_i, and the noise source ρ_N, in addition to the inputs $\boldsymbol{x}$, noisy outputs $\boldsymbol{y}$, and the program space G, it may still be impossible to infer any information about the hidden program. Instead, I explore the problem of synthesizing a program p which *best-fits* the given noisy dataset.

$$\frac{}{[\![c]\!]x \Rightarrow c}\ (\textsc{Constant}) \qquad \frac{}{[\![t]\!]x \Rightarrow x(t)}\ (\textsc{Variable})$$

$$\frac{[\![n_1]\!]x \Rightarrow v_1 \quad [\![n_2]\!]x \Rightarrow v_2 \quad \ldots \quad [\![n_k]\!]x \Rightarrow v_k}{[\![f(n_1, n_2, \ldots n_k)]\!]x \Rightarrow f(v_1, v_2, \ldots v_k)}\ (\textsc{Function})$$

Figure 2-1: Execution semantics for program p.

2.2 Noisy Program Synthesis as an Optimization Problem

I next formalize the problem of synthesizing the *best-fit* program as an optimization problem, given a space of candidate programs and a noisy dataset.

2.2.1 Domain Specific Languages

I first define the set of programs the synthesis process will consider, how inputs to the program are specified, and the program semantics. I assume programs are specified as parse trees in a **domain-specific language (DSL)** grammar $\mathcal{G}$. Internal nodes represent function invocations; leaves are constants/0-arity symbols in the DSL. Given a program p and an input x, $[\![p]\!]x$ denotes the output of p on input x ($[\![.]\!]$ is defined in Figure 2-1).

All valid programs are defined by a DSL grammar $\mathcal{G} = (T, N, P, s_0)$ where:

- T is a set of terminal symbols. These may include constants and symbols which may change value depending on the input x.
- N is the set of nonterminals that represent subexpressions in our DSL.
- P is the set of production rules of the form $s \to f(s_1, \ldots, s_n)$, where f is a built-in function in the DSL and $s, s_1, \ldots, s_n$ are non-terminals in the grammar.
- $s_0 \in N$ is the start non-terminal in the grammar.

I assume that we are given a black box implementation of each built-in function f in the DSL. In general, all techniques explored within this thesis can be generalized to any DSL which can be specified within the above framework. This is a standard way of specifying DSLs in the program synthesis literature [22, 42].

2.2.2 Loss Functions

Given a noisy dataset $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$ and a program p, a **Loss Function** $\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y})$ measures how incorrect the program is with respect to the given dataset. Formally, a loss function $\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y})$ maps the noise free outputs $p[\boldsymbol{x}]$ and noisy outputs $\boldsymbol{y}$ to a non-negative real number or ∞.

Definition 1. $0/1$ **Loss Function:** *The* $0/1$ *loss function* $\mathcal{L}_{0/1}(p[\boldsymbol{x}], \boldsymbol{y})$ *counts the number of input-output examples where* p *does not agree with the dataset* $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$*:*

$$\mathcal{L}_{0/1}(p[\boldsymbol{x}], \boldsymbol{y}) = \sum_{i=1}^{|\boldsymbol{x}|} 1 \quad \text{if} \quad (y_i \neq [\![p]\!] x_i) \quad \text{else} \quad 0$$

Definition 2. $0/\infty$ **Loss Function:** *The* $0/\infty$ *loss function* $\mathcal{L}_{0/\infty}(p[\boldsymbol{x}], \boldsymbol{y})$ *is 0 if* p *matches all outputs in the dataset* $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$ *and* ∞ *otherwise:*

$$\mathcal{L}_{0/\infty}(p[\boldsymbol{x}], \boldsymbol{y}) = 0 \quad \text{if} \quad \boldsymbol{y} = p[\boldsymbol{x}] \quad \text{else} \quad \infty$$

Definition 3. Damerau-Levenshtein (DL) Loss Function: *The DL loss function* $\mathcal{L}_{DL}(p, \mathcal{D})$ *uses the Damerau-Levenshtein metric [10] to measure the distance between the output from the synthesized program and the corresponding output in the noisy dataset:*

$$\mathcal{L}_{DL}(p[\boldsymbol{x}], \boldsymbol{y}) = \sum_{i=1}^{|\boldsymbol{x}|} L_{[\![p]\!] x_i, y_i}\Big(\left|[\![p]\!] x_i\right|, \left|y_i\right| \Big)$$

where, $L_{a,b}(i, j)$ *is the Damerau-Levenshtein metric [10].*

This metric counts the number of single character deletions, insertions, substitutions, or transpositions required to convert one text string into another. Because more than 80% of all human misspellings are reported to be captured by a single one of these four operations [10], the DL loss function may be appropriate for computations that work with human-provided text input-output examples.

2.2.3 Regularizer

Given a program p^*, space of programs G, and inputs $\boldsymbol{x}$, a **Regularizer** $\mathcal{R}(\boldsymbol{x}, p^*[\boldsymbol{x}])$ assigns a weight based on the probability that a random program $p \in G$ will produce

outputs $p^*[\boldsymbol{x}]$ on inputs $\boldsymbol{x}$ ($p[\boldsymbol{x}] = p^*[\boldsymbol{x}]$). Formally a regularizer $\mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}])$ maps the noise-free outputs $p[\boldsymbol{x}]$ and inputs $\boldsymbol{x}$ to a non-negative real number or ∞.

Definition 4. Uniform Regularizer: *The uniform regularizer $\mathcal{R}_U(\boldsymbol{x}, p[\boldsymbol{x}])$ assigns a uniform weight to all programs $p \in G$, i.e.,*

$$\mathcal{R}_U(\boldsymbol{x}, p[\boldsymbol{x}]) = 1$$

Definition 5. Subspace Regularizer: *Given a subset of programs $A \subseteq G$, a subspace regularizer $\mathcal{R}_A(\boldsymbol{x}, \boldsymbol{z})$ returns 1 if there exists a program in A which maps $\boldsymbol{x}$ to $\boldsymbol{z}$, else infinity. Formally,*

$$\mathcal{R}_A(\boldsymbol{x}, \boldsymbol{z}) = 1 \text{ if } \exists p \in A. p[\boldsymbol{x}] = \boldsymbol{z} \text{ else } \infty$$

Let G be a space of programs and let $A \subseteq G$ be a subset of programs in G. Given inputs $\boldsymbol{x}$ and a program p^*, the regularizer $\mathcal{R}_A(\boldsymbol{x}, p^*[\boldsymbol{x}])$ is finite, if and only if, there exists a program $p \in A$, such that, $p[\boldsymbol{x}] = p^*[\boldsymbol{x}]$. This regularizer biases the synthesis process to synthesize a program p^*, which given inputs $\boldsymbol{x}$, produces the same outputs as a program $p \in A$, i.e., $\exists p \in A. p^*[\boldsymbol{x}] = p[\boldsymbol{x}]$.

Definition 6. Distribution Regularizer: *Given a space of programs G and a probability distribution π over programs in G, a distribution regularizer $\mathcal{R}_\pi(\boldsymbol{x}, \boldsymbol{z})$ weights the output vector $\boldsymbol{z}$ based on the probability that a random program $p \in G$ will map inputs $\boldsymbol{x}$ to outputs $\boldsymbol{z}$. Formally,*

$$\mathcal{R}_\pi(\boldsymbol{x}, \boldsymbol{z}) = -\log \pi(G_{\boldsymbol{x},\boldsymbol{z}}) \text{ where } G_{\boldsymbol{x},\boldsymbol{z}} = \{p \in G \mid p[\boldsymbol{x}] = \boldsymbol{z}\}$$

Given prior probability π of programs in G, the distribution regularizer allows us to bias the synthesis process towards programs which produce more likely outputs.

2.2.4 Complexity Measure

Given a program p, a **Complexity Measure** $C(p)$ ranks programs independent of the input-output examples in the dataset $\mathcal{D}$. This measure is used to trade off per-

formance on the noisy dataset vs. complexity of the synthesized program. Formally, a complexity measure is a function $C(p)$ that maps each program p expressible in the given DSL G to a real number. The following $\text{Cost}(p)$ complexity measure computes the complexity of given program p represented as a parse tree recursively as follows:

$$\begin{aligned} \text{Cost}(t) &= \text{cost}(t) \\ \text{Cost}(f(e_1, e_2, \ldots e_k)) &= \text{cost}(f) + \sum_{i=1}^{k} \text{Cost}(e_i) \end{aligned}$$

where t and f are terminals and built-in functions in my DSL respectively. Setting $\text{cost}(t) = \text{cost}(f) = 1$ delivers a complexity measure $\text{Size}(p)$ that computes the size of p.

2.2.5 Objective Function

Given a noisy dataset $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$, loss function $\mathcal{L}$, regularizer $\mathcal{R}$, and a complexity measure C, an **Objective Function** $U(\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(p[\boldsymbol{x}], \boldsymbol{x}), C(p))$ combines the loss, regularizer weight, and complexity to assign a single combine weight to a program p.

Definition 7. Tradeoff Objective Function: *Given tradeoff parameters $\lambda, \gamma > 0$, the tradeoff objective function U_T is a linear combination of the loss, regularizer weight, and the complexity measure, weighted using parameters λ and γ. Formally,*

$$U_T(\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(p[\boldsymbol{x}], \boldsymbol{x}), C(p)) = \mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}) + \lambda \mathcal{R}(p[\boldsymbol{x}], \boldsymbol{x}) + \gamma C(p)$$

This objective function trades the loss of the synthesized program off against the regularization weight and the complexity of the synthesized program. Similarly to how regularization can prevent a machine learning model from overfitting noisy data by biasing the training algorithm to pick a simpler model, the tradeoff objective function may prevent the algorithm from synthesizing a program which overfits the data by biasing it to pick a simpler program (based on the complexity measure).

Definition 8. Lexicographic Objective Function: *Given a tradeoff parameter $\lambda > 0$, a lexicographic objective function $U_L(\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(p[\boldsymbol{x}], \boldsymbol{x}), C(p)) = \langle l, c \rangle$,*

where $l = \mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}) + \lambda\mathcal{R}(p[\boldsymbol{x}], \boldsymbol{x})$ *and* $c = C(p)$*, maps* l *and* c *into a lexicographically ordered space, i.e.,* $\langle l_1, c_1\rangle < \langle l_1, c_2\rangle$ *if and only if either* $l_1 < l_2$ *or* $l_1 = l_2$ *and* $c_1 < c_2$*.*

This objective function first minimizes the combination of loss function and regularizer, then the complexity.

2.2.6 Optimization Problem

Given a set of programs G (specified using a DSL), noisy dataset $\mathcal{D}$, a loss function $\mathcal{L}$, a regularizer $\mathcal{R}$, a complexity measure C, and an objective function U, a program p^* is the *best-fit* program if p^* minimizes the objective function, i.e.,

$$p^* = \arg\min_{p \in G} U(\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(p[\boldsymbol{x}], \boldsymbol{x}), C(p))$$

I introduce synthesis algorithms which synthesize the *best-fit* program, given a DSL, noisy dataset, loss function, regularizer, complexity measure, and an objective function, in Chapters 3 and 6. In Chapter 4, I present experimental results which showcase the ability of my synthesis algorithms to synthesize a program p^*, which is equivalent to the hidden program which generated the noisy dataset. In Chapter 5, I make connections between the optimization based framework and the process which generated the noisy dataset. These connections allow us to quantify the probability of a synthesis algorithm synthesizing a correct program. They also allow us to pick suitable loss functions, regularizers, and complexity measures, which can improve our chances of synthesizing a correct program.

Chapter 3

Synthesis Using Tree-Automata

I next introduce a new synthesis algorithm which uses the loss-based framework to synthesize programs over noisy datasets. This algorithm builds upon the concept of a finite tree automaton [8] to compactly represent programs of a given DSL. The finite tree automaton also allows us to cluster programs based on their loss value. This technique builds upon a prior programming-by-example noise-free synthesis algorithm [43].

3.1 Preliminaries

I first review finite tree automata (FTA) and the FTA-based programming-by-example solution introduced in [43].

3.1.1 Finite Tree Automata

Finite tree automata are a type of state machine which accept trees rather than strings. These machines describe a regular language over trees.

Definition 9 (FTA). *A (bottom-up) finite tree automaton (FTA) over alphabet F is a tuple $\mathcal{A} = (Q, F, Q_f, \Delta)$ where Q is a set of states, $Q_f \subseteq Q$ is the set of accepting states and Δ is a set of transitions of the form $f(q_1, \ldots, q_k) \to q$ where $q, q_1, \ldots q_k$ are states, $f \in F$.*

Every symbol f in alphabet F has an associated arity. The set $F_k \subseteq F$ is the set of all k-arity symbols in F. 0-arity terms t in F are viewed as single node trees

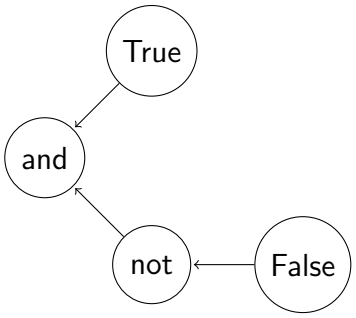

Figure 3-1: Tree for formula $\mathsf{and}(\mathsf{True}, \mathsf{not}(\mathsf{False}))$

(leaves of trees). t is accepted by an FTA if we can rewrite t to some state $q \in Q_f$ using rules in Δ. The language of an FTA $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, corresponds to the set of all ground terms accepted by $\mathcal{A}$.

Example 1. *Consider the tree automaton $\mathcal{A}$ defined by states $Q = \{q_T, q_F\}$, $F_0 = \{\mathsf{True}, \mathsf{False}\}$, $F_1 = \mathsf{not}$, $F_2 = \{\mathsf{and}\}$, final states $Q_f = \{q_T\}$ and the following transition rules Δ:*

$$\begin{array}{cccc} \mathsf{True} \to q_T & \mathsf{False} \to q_F & \mathsf{not}(q_T) \to q_F & \mathsf{not}(q_F) \to q_T \\ \mathsf{and}(q_T, q_T) \to q_T & \mathsf{and}(q_F, q_T) \to q_F & \mathsf{and}(q_T, q_F) \to q_F & \mathsf{and}(q_F, q_F) \to q_F \\ \mathsf{or}(q_T, q_T) \to q_T & \mathsf{or}(q_F, q_T) \to q_T & \mathsf{or}(q_T, q_F) \to q_T & \mathsf{or}(q_F, q_F) \to q_F \end{array}$$

The above tree automaton accepts all propositional logic formulas over True and False which evaluate to True. Figure 3-1 presents the tree for the formula $\mathsf{and}(\mathsf{True}, \mathsf{not}(\mathsf{False}))$.

3.1.2 Concrete Finite Tree Automata

I review the technique, introduced by [43], which uses finite tree automata to solve synthesis tasks over a broad class of DSLs. Given a DSL and a set of input-output examples, a *Concrete Finite Tree Automaton* **(CFTA)** is a tree automaton which accepts all trees representing DSL programs consistent with the input-output examples and nothing else. The states of the FTA correspond to concrete values and the transitions are obtained using the semantics of the DSL constructs.

Given input-output examples $(\boldsymbol{x}, \boldsymbol{y})$ and DSL G, the rules for constructing a CFTA is presented in Figure 3-2. The alphabet of the CFTA contains built-in functions

$$\frac{t \in T, \boldsymbol{z} = \langle [\![t]\!] x_1, \ldots [\![t]\!] x_n \rangle}{q_t^{\boldsymbol{z}} \in Q} \text{ (Term)} \quad \frac{q_{s_0}^{\boldsymbol{y}} \in Q}{q_{s_0}^{\boldsymbol{y}} \in Q_f} \text{ (Final)}$$

$$\frac{\begin{array}{c} s \leftarrow f(s_1, \ldots s_k) \in P, \{q_{s_1}^{\boldsymbol{z}_1}, \ldots q_{s_k}^{\boldsymbol{z}_k}\} \subseteq Q, \\ \boldsymbol{z} = \langle z_1, \ldots z_n \rangle, z_i = [\![f(\boldsymbol{z}_{1,i}, \ldots \boldsymbol{z}_{k,i})]\!] \end{array}}{q_s^{\boldsymbol{z}} \in Q, f(q_{s_1}^{\boldsymbol{z}_1}, \ldots q_{s_k}^{\boldsymbol{z}_k}) \rightarrow q_s^{\boldsymbol{z}} \in \Delta} \text{ (Prod)}$$

Figure 3-2: Rules for constructing a CFTA $\mathcal{A} = (Q, Q_f, \Delta)$ given inputs $\boldsymbol{x} = \langle x_1, \ldots x_n \rangle$ and DSL $G = (T, N, P, s_0)$.

within the DSL. The states in the CFTA are of the form $q_s^{\boldsymbol{z}}$, where s is a symbol (terminal or non-terminal) in G and $\boldsymbol{z}$ is a vector of concrete values. The existence of state $q_s^{\boldsymbol{z}}$ implies that there exists a partial program which can map inputs $\boldsymbol{x}$ to concrete values $\boldsymbol{z}$. Similarly, the existence of transition $f(q_{s_1}^{\boldsymbol{z}_1}, q_{s_2}^{\boldsymbol{z}_2} \ldots q_{s_k}^{\boldsymbol{z}_k}) \rightarrow q_s^{\boldsymbol{z}}$ implies $f(\boldsymbol{z}_{1,j}, \boldsymbol{z}_{2,j} \ldots \boldsymbol{z}_{k,j}) = z_j$, for all $j \in [1, n]$ ($z_{i,j}$ denotes the j^{th} element of vector $\boldsymbol{z}_i$).

The Term rule states that if we have a terminal t (either a constant in the given language or input symbol x), execute it with the input x_i and construct a state $q_t^{\boldsymbol{z}}$ (where $\boldsymbol{z}_i = [\![t]\!] x_i$). The Prod rule states that, if we have a production rule $f(s_1, s_2, \ldots s_k) \rightarrow s \in \Delta$, and there exists states $q_{s_1}^{\boldsymbol{z}_1}, q_{s_2}^{\boldsymbol{z}_2} \ldots q_{s_k}^{\boldsymbol{z}_k} \in Q$, then we also have state $q_s^{\boldsymbol{z}}$ in the FTA and a transition $f(q_{s_1}^{\boldsymbol{z}_1}, q_{s_2}^{\boldsymbol{z}_2}, \ldots q_{s_k}^{\boldsymbol{z}_k}) \rightarrow q_s^{\boldsymbol{z}}$. The CFTA Final rule (Figure 3-2) marks the state $q_{s_0}^{\boldsymbol{y}}$ with start symbol s_0 as accepting state.

The language of the CFTA constructed from Figure 3-2 is exactly the set of parse trees of DSL programs that are consistent with the given input-output examples (i.e., maps inputs $\boldsymbol{x}$ to outputs $\boldsymbol{y}$).

In general, the rules presented in Figure 3-2 may result in a CFTA which has infinitely many states. To control the size of the resulting CFTA, we do not add a new state within the constructed CFTA if the smallest tree accepted by this state is larger than a given threshold d. This results in a CFTA which accepts all programs which are consistent with the input-output examples but are smaller than the given threshold (it can accept some programs which are larger than the given threshold but it never accepts a program which is inconsistent with the given input-output examples). This is standard practice in the synthesis literature [42, 30].

1: **procedure** SYNTHESIZE($\mathcal{D}, G$)
input: noisy dataset $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$, DSL G.
output: A program p^*, such that, $p^* \in \arg\min_{p \in G} U(\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}]), C(p))$
2: $(Q, Q_f, \Delta) := \mathsf{ConstructFTA}(\boldsymbol{x}, G)$;
3: **for** $q \in Q_f$ **do**
4: $p, c := \mathsf{LeastComplex}(q, (Q, Q_f, \Delta), G)$;
5: $P[q] := p$;
▷ Least complex program for a given accepting state.
6: $q^{\boldsymbol{v}}_{s_0} \in Q_f$; $q^* := q^{\boldsymbol{v}}_{s_0}$; $\boldsymbol{v}^* := \boldsymbol{v}$;
7: **for** $q^{\boldsymbol{v}}_{s_0} \in Q_f$ **do**
8: **if** $U(\mathcal{L}(\boldsymbol{v}^*, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, \boldsymbol{v}^*), C(P[q^*])) \leq U(\mathcal{L}(\boldsymbol{v}, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, \boldsymbol{v}), C(P[q^{\boldsymbol{v}}_{s_0}]))$ **then**
9: $\boldsymbol{v}^* = \boldsymbol{v}$;
10: $q^* := q^{\boldsymbol{v}}_{s_0}$;
▷ q^* is the accepting state which accepts a program which minimizes the objective function.
11: **return** $P[q^*]$;

Figure 3-3: Algorithm for noisy program synthesis using finite tree automaton.

3.2 Synthesis Algorithm

Figure 3-3 presents my algorithm for synthesizing programs over noisy datasets. The algorithm, given a noisy dataset $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$, a DSL G, an objective function U, a loss function $\mathcal{L}$, a regularizer $\mathcal{R}$, and a complexity measure C, synthesizes a program which minimizes the objective function, i.e., the program p^* returned by the algorithm satisfies the following constraint:

$$p^* \in \mathsf{argmin}_{p \in G} U(\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}]), C(p))$$

The algorithm first constructs a finite tree automaton (line 2) based on the rules presented in Figure 3-2 with a single exception. All states attached with the start symbol are added to the set of accepting states, i.e,

$$\frac{q^{\boldsymbol{v}}_{s_0} \in Q}{q^{\boldsymbol{v}}_{s_0} \in Q_f} \text{ (FINAL)}$$

Even states with start symbol s_0 and attached values $\boldsymbol{v}$ not equal to the noisy

dataset outputs $\boldsymbol{y}$ are added to the set of accepting states.

Given an accepting state of the form $q^{\boldsymbol{v}}_{s_0}$, a program $p \in G$ is accepted by the automaton $(Q, \{q^{\boldsymbol{v}}_{s_0}\}, \Delta)$ if and only if $p[\boldsymbol{x}] = \boldsymbol{v}$.

Theorem 1. *Given a dataset $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$, DSL G, and automaton $(Q, Q_f, \Delta) =$ $\mathsf{ConstructFTA}(\boldsymbol{x}, G)$, for all symbols $s \in G$, for all expressions e starting from symbol s (and height less than bound b), there exists a state $q^{\boldsymbol{v}}_s \in Q$, such that, e is accepted by the automaton $(Q, \{q^{\boldsymbol{v}}_s\}, \Delta)$, where $\boldsymbol{v} = [\![e]\!]\boldsymbol{x}$. Given a state $q^{\boldsymbol{v}}_s \in Q$, if the automaton $(Q, \{q^{\boldsymbol{v}}_s\}, \Delta)$ accepts an expression e, then e starts from symbol s, and $[\![e]\!]\boldsymbol{x} = \boldsymbol{v}$.*

Proof. I prove this theorem by using induction over height of the expression e.

Base Case: Height of expression e is 1. This implies s is either x or a constant. According to Var and Const rules (Figure 3-2), there exists a state $q^{\boldsymbol{v}}_e \in Q$ (for terminal t), such that, e is accepted by the automaton $(Q, \{q^{\boldsymbol{v}_t}\}, \Delta)$, where $\boldsymbol{v} = [\![e]\!]\boldsymbol{x}$.

For a terminal t, if state $q^{\boldsymbol{v}}_t \in Q$ and e is accepted by $(Q, \{q^{\boldsymbol{v}_t}\}, \Delta)$, then $e = t$ and $[\![e]\!]\boldsymbol{x} = \boldsymbol{v}$ (Const and Var rules in Figure 3-2).

Induction Hypothesis: For all symbols $s \in G$, for all expressions e starting from symbol s of height less than equal to n (and height less than bound b), there exists a state $q^{\boldsymbol{v}}_s \in Q$, such that, e is accepted by the automaton $(Q, \{q^{\boldsymbol{v}}_s\}, \Delta)$, where $\boldsymbol{v} = [\![e]\!]\boldsymbol{x}$. Given a state $q^{\boldsymbol{v}}_s \in Q$, if the automaton $(Q, \{q^{\boldsymbol{v}}_s\}, \Delta)$ accepts an expression e of height less than equal to n, then e starts form symbol s, and $[\![e]\!]\boldsymbol{x} = \boldsymbol{v}$.

Induction Step: For any symbol s in G, consider an expression $e = f(e_1, \ldots e_k)$ of height equal to $n + 1$, created from production $s \leftarrow f(s_1, \ldots s_k)$. Note the height of expressions $e_1, \ldots e_k$ is less than equal to n, therefore using induction hypothesis, there exists states $q^{\boldsymbol{v}_1}_{s_1}, \ldots q^{\boldsymbol{v}_k}_{s_k} \in Q$, such that e_i is accepted by automaton $(Q, \{q^{\boldsymbol{v}_i}_{s_i}\}, \Delta)$, where $\boldsymbol{v}_i = \langle [\![e_i]\!]x_1, \ldots [\![e_i]\!]x_n \rangle$.

According to Prod rule (Figure 3-2), there exists a state $q^{\boldsymbol{v}}_s \in Q$, where

$$\boldsymbol{v} = \langle [\![e]\!]x_1, \ldots [\![e]\!]x_n \rangle$$

and e is accepted by $(Q, \{q^{\boldsymbol{v}}_s\}, \Delta)$.

```
1: procedure LeastComplex(q_s^v, A, G, M)
2:     if q ∈ keys(M) then
3:         return M[q_s^v]
4:     if s ∈ T then
5:         return t, cost(t);
6:     M[q_s^v] := ⊥, ∞;
7:     c* := ∞;
8:     for f(q_{s1}^{v1}, ..., q_{sk}^{vk}) → q_s^v ∈ Δ  do
9:         for i := 1 ... k do
10:            e_i, c_i := LeastComplex(q_{si}^{vi}, A, G, M);
11:        e := f(e_1, ... e_k);
12:        c := cost(f) + Σ_{i=1}^{n} c_i;
13:        if c < c* then
14:            e* := e
15:            c* := c;
16:    M[q_s^v] := e*, c*;
17:    return e*, c*;
18: procedure LeastComplex(q_s^v, A, G)
    input: State q_s^v, FTA A = (Q, Q_f, Δ), and DSL G.
    input: Recursively defined complexity measure cost.
    output: Least complex expression e* and it's complexity c*.
19:    M := ∅                                          ▷ Empty map
20:    return LeastComplex(q_f, A, G, M)
```

Figure 3-4: Algorithm for synthesizing a least complex program for automaton $\mathcal{A}$, DSL G, and state $q_s^{\boldsymbol{v}}$.

Therefore, by induction, for all symbols s in G, for all expressions e starting from symbol s (and height less than bound b), there exists a state $q_s^{\boldsymbol{v}} \in Q$, such that, e is accepted by the automaton $(Q, \{q_s^{\boldsymbol{v}}\}, \Delta)$, where $\boldsymbol{v} = \langle [\![e]\!]x_1, \ldots [\![e]\!]x_n \rangle$.

Consider a state $q_s^{\boldsymbol{v}}$, which accepts an expression $e = f(e_1, \ldots e_k)$ of height $n+1$. This implies, for all $i \in [1, k]$, there exists a state $q_{s_i}^{\boldsymbol{v}_i}$ which accepts expression e_i. Using induction hypothesis, e_i starts from symbol s_i and $[\![e_i]\!]\boldsymbol{x} = \boldsymbol{v}$.

According to Prod rule (Figure 3-2), there exists a production rule $s \leftarrow f(s_1, \ldots s_k)$. Therefore, e starts from symbol s and $\boldsymbol{v} = [\![e]\!]\boldsymbol{x}$. □

Given this finite tree automaton (Q, Q_f, Δ), the algorithm finds the least complex program (i.e., the program which minimizes the complexity measure) for each accepting state $q \in Q_f$ (line 3-4). I present the pseudo code for LeastComplex algorithm in

Figure 3-4.

Given an accepting $q_{s_0}^{\boldsymbol{v}}$, $P[q_{s_0}^{\boldsymbol{v}}]$ is the least complex program which maps input vector $\boldsymbol{x}$ to outputs $\boldsymbol{v}$, i.e.,

$$P[q_{s_0}^{\boldsymbol{v}}] \in \mathsf{argmin}_{p \in G[\boldsymbol{x} \to \boldsymbol{v}]} C(p)$$

where $G[\boldsymbol{x} \to \boldsymbol{v}] = \{p \mid p \ \in G, p[\boldsymbol{x}] = \boldsymbol{v}\}$.

Theorem 2. *Given a DSL G, a dataset $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$, $(Q, Q_f, \Delta) = \mathsf{ConstructFTA}(\boldsymbol{x}, G)$, for all states $q_{s_0}^{\boldsymbol{v}} \in Q_f$, if $p^*, c = \mathsf{LeastComplex}(q_{s_0}^{\boldsymbol{v}}, (Q, Q_f, \Delta), G)$, then*

$$p^* \in \mathsf{argmin}_{p \in G[\boldsymbol{x} \to \boldsymbol{v}]} C(p)$$

where $G[\boldsymbol{x} \to \boldsymbol{v}] = \{p \mid p \ \in G, p[\boldsymbol{x}] = \boldsymbol{v}\}$.

Proof. From pseudo code of $\mathsf{LeastComplex}$, the procedure returns the small cost tree accepted by $(Q, \{q_{s_0}^{\boldsymbol{v}}\}, \Delta)$. From Theorem 1,

$$p^* \in \mathsf{argmin}_{p \in G[\boldsymbol{x} \to \boldsymbol{v}]} C(p)$$

where $G[\boldsymbol{x} \to \boldsymbol{v}] = \{p \mid p \ \in G, p[\boldsymbol{x}] = \boldsymbol{v}\}$. □

The algorithm then finds an accepting state $q_{s_0}^{\boldsymbol{v}^*} \in Q_f$, such that, for all accepting states $q_{s_0}^{\boldsymbol{v}} \in Q_f$,

$$U(\mathcal{L}(\boldsymbol{v}^*, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, \boldsymbol{v}^*), C(P[q_{s_0}^{\boldsymbol{v}^*}]) \leq U(\mathcal{L}(\boldsymbol{v}, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, \boldsymbol{v}), C(P[q_{s_0}^{\boldsymbol{v}}])$$

Since, for all programs $p \in G$, there exists an accepting state $q_{s_0}^{\boldsymbol{v}}$, such that, p is accepted by $(Q, \{q_{s_0}^{\boldsymbol{v}}\}, \Delta)$, the following statement is true:

$$U(\mathcal{L}(\boldsymbol{v}, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, \boldsymbol{v}), C(P[q_{s_0}^{\boldsymbol{v}}]) \leq U(\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}]), C(p))$$

Therefore, the program $P[q_{q_{s_0}}^{\boldsymbol{v}^*}]$ is the best-fit program which minimizes the objective

function, i.e.,

$$P[q_{s_0}^{\boldsymbol{v}^*}] \in \mathsf{argmin}_{p \in G} U(\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}]), C(p))$$

Theorem 3. *Given a dataset $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$ and a DSL G, the synthesis algorithm* Synthesize *(Figure 3-3) returns the program which minimizes the objective function.*

Proof. Let $(Q, Q_f, \Delta) = \mathsf{ConstructFTA}(\mathcal{D}, G)$. For all programs $p \in G$ (of height less than the given bound), there exits a state $q_{s_0}^{\boldsymbol{v}} \in Q_f$, such that, $p[\boldsymbol{x}] = \boldsymbol{v}$ (Theorem 1). Let $G_{\boldsymbol{x},\boldsymbol{v}}$ be the subset of programs $p \in G$, such that, $p[\boldsymbol{x}] = \boldsymbol{v}$.

For each state $q_{s_0}^{\boldsymbol{v}}$, LeastComplex returns program $P[q_{s_0}^{\boldsymbol{v}}]$, such that, $P[q_{s_0}^{\boldsymbol{v}}] \in \arg\min_{p \in G_{\boldsymbol{x},\boldsymbol{v}}} C(p)$. Note that, this implies

$$P[q_{s_0}^{\boldsymbol{v}}] \in \arg\min_{p \in G_{\boldsymbol{x},\boldsymbol{v}}} U(\mathcal{L}(\boldsymbol{v}, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, \boldsymbol{v}), C(p))$$

Algorithm Synthesize returns program $P[q^*]$, such that,

$$q^* \in \arg\min_{q_{s_0}^{\boldsymbol{v}} \in Q_f} U(\mathcal{L}(\boldsymbol{v}, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, \boldsymbol{v}), C(P[q]))$$

This implies

$$P[q^*] \in \arg\min_{p \in \bigcup_{q_{s_0}^{\boldsymbol{v}} \in Q_f} G_{\boldsymbol{x},\boldsymbol{v}}} U(\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}]), C(p))$$

Note that, $\bigcup_{q_{s_0}^{\boldsymbol{v}} \in G} G_{\boldsymbol{x},\boldsymbol{v}} = G$ (Theorem 1). Therefore,

$$P[q^*] \in \arg\min_{p \in G} U(\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}]), C(p))$$

Therefore, Synthesize returns the program which minimizes the objective function. □

3.3 Implementation

With string transformations as a potential application domain, I implemented our tree automaton based technique (in 6k lines of Java code) for a DSL from [42] (Figure 3-5). The DSL supports extracting and concatenating (Concat) substrings of the input string x; each substring is extracted using the SubStr function with a start and end

$$
\begin{array}{rrcl}
\text{String expr} & e & := & \mathsf{Str}(f) \mid \mathsf{Concat}(f, e); \\
\text{Substring expr} & f & := & \mathsf{ConstStr}(s) \mid \mathsf{SubStr}(x, p_1, p_2); \\
\text{Position} & p & := & \mathsf{Pos}(x, \tau, k, d) \mid \mathsf{ConstPos}(k) \\
\text{Direction} & d & := & \mathsf{Start} \mid \mathsf{End};
\end{array}
$$

Figure 3-5: DSL for string transformation, τ is a token, k is an integer, and s is a string constant.

position. A position can either be a constant index (ConstPos) or the start or end of the k^{th} occurrence of the match token τ in the input string (Pos).

Implementation Optimizations: My implementation applies two techniques that constrain the size of the final FTA. First, it bounds the number of recursive applications of the production $e := \mathsf{Concat}(f, e)$ by applying a *bounded scope height threshold* d. This restricts the search space to programs in our DSL which contains a maximum of d concatenations. Since the production $e := \mathsf{Concat}(f, e)$ is the only recursive production rule, the *bounded scope height threshold* restricts our search space to finite set of programs. Second, during construction of the FTA, a state with symbol e is only added to the FTA if the length of the state's output value is not greater than the length of the output string plus one.

3.4 Discussion

This chapter presents a new program synthesis algorithm for synthesizing programs over noisy dataset. This algorithm builds upon a prior technique [43] which uses concrete finite tree automaton (CFTA) to synthesize programs over noise-free datasets. The CFTA constructed by the original technique has a single accepting state. This state is of the form $q_{s_0}^{\boldsymbol{y}}$, where s_0 is the start symbol in the given DSL and $\boldsymbol{y}$ is the noise-free outputs in the noise-free dataset. One can identify that CFTA, in this technique, is an efficient data structure to represent programs which map inputs to the corresponding outputs.

Our technique builds upon this property of CFTA. It builds a CFTA that accepts all programs in the given DSL. If a program is accepted by state $q_{s_0}^{\boldsymbol{z}}$, then that program returns $\boldsymbol{z}$ on the given inputs. For each accepting state, our technique synthesizes the least complex program accepted by that state. Our technique then iterates over all accepting states and, using the least complex program accepted by the accepting

state, synthesizes the program which minimizes the objective function. Our technique generalizes the efficient CFTA data structure from the prior technique to synthesize programs over noisy datasets [42].

Chapter 4

Experimental Results

String transformations have been extensively studied within the programming by example community [20, 30, 35]. I use the SyGuS 2018 benchmark suite [1] to benchmark my technique. This benchmark suite contains a range of string transformation problems, a class of problems that has been extensively studied in past program synthesis projects [20, 30, 35].

I use the size complexity measure Size(p) (Subsection 2.2.4) and the uniform regularizer $\mathcal{R}_U$ (Definition 4) for these experiments.

4.1 Scalability

Before I start exploring the performance of this technique in presence of different noise sources and loss functions for different noisy datasets, I first evaluate the runtime performance and scalability of my technique. I evaluate my implementation by applying it to all problems in the SyGuS 2018 benchmark suite [1]. My technique applies the same loss function and regularizer on each accepting state, independent of the noisy output values. The number of accepting states and the values attached to each accepting state is dependent of the domain specific language and the inputs in the dataset, but is independent of the noisy outputs in the dataset. The performance of my technique is independent of the noise in the noisy dataset. Therefore, for each problem, I use a clean (noise-free) dataset for the problem provided with the benchmark suite to evaluate the runtime performance and scalability of my technique.

I use the lexicographic objective function U_L with the $0/\infty$ loss function, uniform regularizer $\mathcal{R}_U$, and the $c = \text{Size}(p)$ complexity measure. I run each benchmark with bounded scope height threshold $d = 1$, 2, 3, and 4 and record the running time on that benchmark problem and the number of states in the FTA (Section 6.7). I use the bounded scope height threshold d to restrict the search space to a finite set of programs. Given a bounded threshold d, our search space only contains programs contains at most d instances of function Concat. A state with symbol e is only added to the FTA if the length of its output value is not greater than the length of the output string.

Because the running time of my technique does not depend on the specific objective function (except for the time required to evaluate the objective function, which is typically negligible for most objective functions), we anticipate that these results will generalize to other objective functions. All experiments are run on an 3.00 GHz Intel(R) Xeon(R) CPU E5-2690 v2 machine with 512GB memory running Linux 4.15.0. With a timeout limit of 10 minutes and bounded scope height threshold of 4, the implementation is able to solve 64 out of the 108 SyGuS 2018 benchmark problems. For the remaining 48 benchmark problems, either fails to return the correct program or it times out.

Figures 4-1 and 4-2 present my results for SyGuS 2018 benchmarks. There is a row for each benchmark problem. The first column presents the name of the benchmark. The next four columns present results for the technique running with bounded scope height threshold $d = 1$, 2, 3, and 4, respectively. Each column has two subcolumns: the first presents the running time on that benchmark problem (in seconds); the second presents the number of states in the FTA (in thousands of states). An entry X indicates that the implementation terminated but did not synthesize a correct program that agreed with all provided input-output examples. An entry - indicates that the implementation did not terminate.

In general, both the running times and the number of states in the FTA increase as the number of provided input-output examples and/or the bounded height threshold increases. The FTA size sometimes stops increasing as the height threshold increases.

Threshold	1		2		3		4	
Benchmark name	time(sec)	FTA size	time(sec)	FTA size	time(sec)	FTA size	time(sec)	FTA size
bikes	0.16	1.08	0.73	10.56	4.72	56.4	19.83	145.8
bikes-long	0.21	1.02	1.37	9.42	6.04	49.9	26.99	139.35
bikes-long-repeat	0.18	1.02	1.06	9.42	6.03	49.9	27.47	139.35
bikes-short	0.15	1.08	0.79	10.56	3.98	56.4	18.62	145.8
dr-name	X	X	7.54	107.5	107.18	1547.2	-	-
dr-name-long	X	X	17.4	70.28	300.9	1077.6	-	-
dr-name-long-repeat	X	X	19.15	70.28	301.3	1077.6	-	-
dr-name-short	X	X	10.2	107.5	101.5	154.8	-	-
firstname	0.28	1.02	1.46	4.34	4.024	4.33	3.97	4.34
firstname-long	1.72	1.04	12.03	4.36	39.08	4.36	41.22	4.36
firstname-long-repeat	1.64	1.04	13.96	4.36	42.4	4.36	43.1	4.36
firstname-short	0.26	1.02	1.47	4.37	3.93	4.34	3.9	4.34
initials	X	X	X	X	8.7	42.3	30.4	42.34
initials-long	X	X	X	X	86.44	42.36	376.56	42.36
initials-long-repeat	X	X	X	X	86.23	42.36	386.25	42.36
initials-short	X	X	X	X	8.92	42.34	31.72	42.34
lastname	0.43	2.56	4.78	28.3	27.29	208.35	159.41	741.44
lastname-long	1.93	1.37	15.1	11.34	112.04	50.81	485.98	50.8
lastname-long-repeat	1.85	1.37	18.35	11.34	113.36	50.81	486.35	50.8
lastname-short	0.6	2.56	3.07	28.3	28.3	208.35	160.54	741.44
name-combine	X	X	8.49	269.9	224.074	7485.83	-	-
name-combine-long	X	X	32.28	161.54	-	-	-	-
name-combine-long-repeat	X	X	98.46	299	-	-	-	-
name-combine-short	X	X	6.5	269.9	207.86	7485.83	-	-
name-combine-2	X	X	X	X	63.490	855.34	-	-
name-combine-2-long	X	X	X	X	591.6	851.44	-	-
name-combine-2-long-repeat	X	X	X	X	592.0	851.44	-	-
name-combine-2-short	X	X	X	X	57.26	855.34	-	-
name-combine-3	X	X	X	X	43.082	911.53	527.29	8104.7
name-combine-3-long	X	X	X	X	193.42	649.13	-	-
name-combine-3-long-repeat	X	X	X	X	192.81	649.13	-	-
name-combine-3-short	X	X	X	X	42.266	911.53	526.13	8104.7
name-combine-4	X	X	X	X	X	X	-	-
name-combine-4-long	X	X	X	X	-	-	-	-
name-combine-4-long-repeat	X	X	X	X	-	-	-	-
name-combine-4-short	X	X	X	X	X	X	-	-
reverse-name	X	X	6.9	269.9	217.19	7495.9	-	-
reverse-name-long	X	X	29.55	161.53	-	-	-	-
reverse-name-long-repeat	X	X	27.6	161.53	-	-	-	-
reverse-name-short	X	X	6.84	269.9	228.24	7485.8	-	-

Figure 4-1: Runtime and FTA size for SyGuS 2018 benchmarks.

I attribute this phenomenon to the application of a search space pruning technique that terminates the recursive application of the production $e := \mathsf{Concat}(f, e)$; when the generated string becomes longer than the current output string — in this case any resulting synthesized program will produce an output that does not match the output in the dataset.

I compare my technique with a previous technique that uses FTAs to solve program synthesis problems [42]. This previous technique requires clean data and only synthesizes programs that agree with all input-output examples in the dataset. My technique builds FTAs with similar structure, with additional overhead coming from the evaluation of the objective function. I obtained the implementation of the technique presented in [42] and ran this implementation on all benchmarks in the SyGuS 2018 benchmark suite. If both of these techniques construct the CFTA on all examples at the same time, the running times of my implementation and this previous implementation are comparable.

Threshold	1		2		3		4	
Benchmark name	time(sec)	FTA size	time(sec)	FTA size	time(sec)	FTA size	time(sec)	FTA size
phone	0.12	0.46	0.47	1.58	0.87	1.58	0.78	1.58
phone-long	0.8	0.46	3.9	1.58	7.79	1.58	32.79	1.58
phone-long-repeat	0.69	0.46	3.29	1.58	7.76	1.58	43.24	1.58
phone-short	0.12	0.46	0.37	1.58	0.804	1.578	4.97	1.58
phone-1	0.15	0.46	0.44	1.58	0.84	1.58	3.017	1.58
phone-1-long	0.99	0.46	3.8	1.58	8.23	1.58	16.58	1.58
phone-1-long-repeat	0.90	0.46	4.1	1.58	8.42	1.58	17.5	1.58
phone-1-short	0.14	0.46	0.45	1.58	0.8	1.58	1.5	1.58
phone-2	0.13	0.46	0.44	1.58	0.83	1.58	3.176	1.58
phone-2-long	0.64	0.46	2.84	1.58	8.36	1.58	16	1.58
phone-2-long-repeat	0.85	0.46	3.8	1.58	9.83	1.58	17.55	1.58
phone-2-short	0.09	0.46	0.47	1.58	0.83	1.58	2.78	1.58
phone-3	X	X	X	X	X	X	-	-
phone-3-long	X	X	X	X	-	-	-	-
phone-3-long-repeat	X	X	X	X	-	-	-	-
phone-3-short	X	X	X	X	X	X	-	-
phone-4	X	X	X	X	X	X	-	-
phone-4-long	X	X	X	X	-	-	-	-
phone-4-long-repeat	X	X	X	X	-	-	-	-
phone-4-short	X	X	X	X	X	X	-	-
phone-5	0.18	0.23	0.16	0.23	0.11	0.23	0.7	0.23
phone-5-long	1.24	0.23	0.94	0.23	0.75	0.23	4.2	0.23
phone-5-long-repeat	1.27	0.23	1.19	0.23	0.77	0.23	2.96	0.23
phone-5-short	0.17	0.23	0.17	0.23	0.11	0.23	0.9	0.23
phone-6	0.27	0.64	1.38	2.6	2.67	2.61	9.3	2.61
phone-6-long	1.84	0.64	6.48	2.6	24.66	2.61	103.3	2.61
phone-6-long-repeat	2.16	0.64	7.12	2.6	24.69	2.61	143.9	2.61
phone-6-short	0.28	0.64	0.76	2.6	2.27	2.61	11.19	2.61
phone-7	0.24	0.64	1.04	2.6	2.87	2.61	11.141	2.61
phone-7-long	2.6	0.64	7.8	2.6	26.1	2.61	108.1	2.61
phone-7-long-repeat	2.58	0.64	6.68	2.6	26.15	2.61	115.42	2.61
phone-7-short	0.23	0.64	1.13	2.6	3.26	2.61	10.71	2.61
phone-8	0.23	0.64	1	2.6	2.65	2.61	8.51	2.61
phone-8-long	2.33	0.64	7.58	2.6	25.87	2.61	114.54	2.61
phone-8-long-repeat	1.67	0.64	7.7	2.6	25.45	2.61	128.3	2.61
phone-8-short	0.27	0.64	0.97	2.6	2.45	2.61	13.81	2.61
phone-9	X	X	X	X	-	-	-	-
phone-9-long	X	X	X	X	-	-	-	-
phone-9-long-repeat	X	X	X	X	-	-	-	-
phone-9-short	X	X	X	X	-	-	-	-
phone-10	X	X	X	X	-	-	-	-
phone-10-long	X	X	X	X	-	-	-	-
phone-10-long-repeat	X	X	X	X	-	-	-	-
phone-10-short	X	X	-	-	-	-	-	-
univ-1	X	X	-	-	-	-	-	-
univ-1-long	X	X	-	-	-	-	-	-
univ-1-long-repeat	X	X	-	-	-	-	-	-
univ-1-short	X	X	-	-	-	-	-	-
univ-2	X	X	-	-	-	-	-	-
univ-2-long	X	X	-	-	-	-	-	-
univ-2-long-repeat	X	X	-	-	-	-	-	-
univ-2-short	X	X	-	-	-	-	-	-
univ-3	X	X	-	-	-	-	-	-
univ-3-long	X	X	-	-	-	-	-	-
univ-3-long-repeat	X	X	-	-	-	-	-	-
univ-3-short	X	X	-	-	-	-	-	-
univ-4	X	X	-	-	-	-	-	-
univ-4-long	X	X	-	-	-	-	-	-
univ-4-long-repeat	X	X	-	-	-	-	-	-
univ-4-short	X	X	-	-	-	-	-	-
univ-5	X	X	-	-	-	-	-	-
univ-5-long	X	X	-	-	-	-	-	-
univ-5-long-repeat	X	X	-	-	-	-	-	-
univ-5-short	X	X	-	-	-	-	-	-
univ-6	X	X	-	-	-	-	-	-
univ-6-long	X	X	-	-	-	-	-	-
univ-6-long-repeat	X	X	-	-	-	-	-	-
univ-6-short	X	X	-	-	-	-	-	-

Figure 4-2: Runtimes and FTA sizes for SyGuS 2018 benchmarks.

An implementation of their technique also makes an additional optimization. Their algorithm only constructs the FTA on a subset of input-output examples. Initially, this subset only contains a single example. Their algorithm builds the CFTA on this subset and synthesizes the simplest program accepted by this CFTA. If this program satisfies all examples in the dataset, the algorithm returns this program. If there

Benchmark name	Programming-by-example technique	Without optimization	Our technique
bikes	1.422	4.98	4.72
bikes-long	1.099	5.8	6.04
bikes-long-repeat	1.114	6.48	6.03
bikes_small	1.083	4.9	3.98
dr-name	15.438	121.8	107.18
dr-name-long	10.645	285.8	300.9
dr-name-long-repeat	10.342	280.6	301.3
dr-name_small	15.355	120.3	101.5
firstname	1.822	3.96	4.024
firstname-long	1.974	38.03	39.08
firstname-long-repeat	2.014	44.84	42.4
firstname_small	1.98	4.135	3.93
initials	0.926	8.84	8.7
initials-long	0.954	99.73	86.44
initials-long-repeat	0.854	82.362	86.23
initials_small	1.024	9.024	8.92
lastname	12.128	28.7	27.29
lastname-long	12.227	103.4	112.04
lastname-long-repeat	11.389	103.5	113.36
lastname_small	11.556	30.54	28.3
name-combine	26.267	201.24	224.074
name-combine-long	24.421	-	-
name-combine-long-repeat	63.194	-	-
name-combine_short	28.125	196.21	207.86
name-combine-2	10.076	68.91	63.49
name-combine-2-long	10.144	594.13	591.6
name-combine-2-long-repeat	10.885	595.6	592
name-combine-2_short	11.126	66.723	57.26
name-combine-3	13.113	45.16	43.082
name-combine-3-long	13.857	217.66	193.42
name-combine-3-long-repeat	14.358	180.66	192.81
name-combine-3_short	13.873	49.83	42.26
name-combine-4	-	-	-
name-combine-4-long	-	-	-
name-combine-4-long-repeat	-	-	-
name-combine-4_short	-	-	-
phone	0.211	1.014	0.87
phone-long	0.211	8.46	7.79
phone-long-repeat	0.207	6.64	7.76
phone_short	0.203	0.924	0.804
phone-1	0.205	0.967	0.84
phone-1-long	0.21	8.59	8.23
phone-1-long-repeat	0.206	7.89	8.42
phone-1_short	0.211	0.95	0.8
phone-2	0.203	0.954	0.83
phone-2-long	0.206	8.73	8.36
phone-2-long-repeat	0.203	8.7	9.83
phone-2_short	0.206	0.95	0.83
phone-3	-	-	-
phone-3-long	-	-	-
phone-3-long-repeat	-	-	-
phone-3_short	-	-	-
phone-4	-	-	-
phone-4-long	-	-	-
phone-4-long-repeat	-	-	-
phone-4_short	-	-	-
phone-5	0.323	0.124	0.11
phone-5-long	0.321	0.737	0.75
phone-5-long-repeat	0.308	0.696	0.77
phone-5_short	0.367	0.12	0.11
phone-6	0.315	3.097	2.67
phone-6-long	0.317	26.522	24.66
phone-6-long-repeat	0.303	22.97	24.69
phone-6_short	0.324	2.961	2.27
phone-7	0.321	2.889	2.87
phone-7-long	0.331	26.85	26.1
phone-7-long-repeat	0.337	27.08	26.15
phone-7_short	0.311	3.225	3.26
phone-8	0.327	2.93	2.65
phone-8-long	0.329	27	25.87
phone-8-long-repeat	0.325	27.524	25.45
phone-8_short	0.305	2.92	2.45

Figure 4-3: Runtimes for our noisy program synthesis technique vs CFTA based programming by example technique over SyGuS 2018 benchmarks.

exists an example, on which the output of this program is not equal to the output in the dataset, the algorithm adds this example to the subset. Then the algorithm repeats this process until it synthesizes a program which satisfies all input-output

Benchmark name	Programming-by-example technique	Without optimization	Our technique
phone-9	-	-	-
phone-9-long	-	-	-
phone-9-long-repeat	-	-	-
phone-9_short	-	-	-
phone-10	-	-	-
phone-10-long	-	-	-
phone-10-long-repeat	-	-	-
phone-10_short	-	-	-
reverse-name	29.211	206.53	217.19
reverse-name-long	25.78	-	-
reverse-name-long-repeat	24.654	-	-
reverse-name_short	25.418	202.85	228.24
univ_1	-	-	-
univ_1-long	-	-	-
univ_1-long-repeat	-	-	-
univ_1_short	-	-	-
univ_2	-	-	-
univ_2-long	-	-	-
univ_2-long-repeat	-	-	-
univ_2_short	-	-	-
univ_3	-	-	-
univ_3-long	-	-	-
univ_3-long-repeat	-	-	-
univ_3_short	-	-	-
univ_4	-	-	-
univ_4-long	-	-	-
univ_4-long-repeat	-	-	-
univ_4_short	-	-	-
univ_5	-	-	-
univ_5-long	-	-	-
univ_5-long-repeat	-	-	-
univ_5_short	-	-	-
univ_6	-	-	-
univ_6-long	-	-	-
univ_6-long-repeat	-	-	-
univ_6_short	-	-	-

Figure 4-4: Runtimes for our noisy program synthesis technique vs CFTA based programming by example technique over SyGuS 2018 benchmarks.

examples in the dataset.

Figures 4-3 and 4-4 present the runtime (in sec) of CFTA based programming-by-example technique presented in [42], their technique without the additional optimization, and our technique. Compared to our technique, their technique has a median speedup of 8.5x. Without the additional optimization, the performance of our technique and their technique is similar.

4.2 Noisy Data Sets, Character Deletion

I next present results for my implementation running on small (few input-output examples) datasets with character deletions. I use a noise source that cyclically deletes a single character from each output in the dataset in turn, starting with the first character, proceeding through the output positions, then wrapping around to the first character again. I apply this noise source to corrupt every output in the dataset. To construct a noisy dataset with k correct (uncorrupted) outputs, I do not apply the noise source to the last k outputs in the dataset.

I exclude all benchmarks that do not terminate within the time limit at height bound 3. Compared to other bounds explored in our scalability experiments, my technique with height bound 3 is able to synthesize the correct program for the maximum number of benchmarks within the given timeout (10 minutes for each benchmark). For each remaining benchmark I use my implementation and the generated corrupted datasets to determine the minimum number of correct outputs in the corrupted dataset required for the implementation to produce a correct program that matches the original hidden clean dataset on all input-output examples. I consider three loss functions: the 0/1 loss function and DL loss function and the following 1-Delete loss function, which is designed to work with noise sources that delete a single character from the output stream:

Definition 10. 1-Delete Loss Function: *The 1-Delete loss function* $\mathcal{L}_{1D}(\boldsymbol{x}, \boldsymbol{y})$*, for each example, assigns loss 0 if the outputs from the synthesized program and the dataset match exactly, 1 if a single deletion enables the output from the synthesized program to match the output from the dataset, and* ∞ *otherwise:*

$$\mathcal{L}_{1D}(\langle z_1, \ldots z_n \rangle, \langle y_1, \ldots y_n \rangle) = \sum_{i=1}^{n} L_{1D}(z_i, y_i), \text{ where}$$

$$L_{1D}(z, y) = \begin{cases} 0 & z = y \\ 1 & a \cdot w \cdot b = z \wedge a \cdot b = y \wedge |w| = 1 \\ \infty & \textit{otherwise} \end{cases}$$

I use the lexicographic objective function U_L with $c = \text{Size}(p)$ as the complexity measure and bounded scope height threshold $d = 4$.

Figure 4-5 summarizes a subset of the results. The Data Set Size Column presents the total number of input-output examples in the corrupted dataset. The next three columns, 1-Delete, DL, and 0/1, present the minimum number of correct (uncorrupted) input-output examples required for the technique to synthesize a correct program (that agrees with the original hidden clean dataset on all input-output examples) using the 1-Delete, DL, and 0/1 loss functions, respectively.

Benchmark	Data set size	Number of required correct examples		
		1-Delete	DL	0/1
bikes	6	0	0	3
dr-name	4	0	0	2
firstname	4	0	0	2
lastname	4	0	1	1
initials	4	0	2	2
reverse-name	6	0	0	2
name-combine	6	0	0	2
name-combine-2	4	0	0	2
name-combine-3	6	0	0	2
phone	6	0	2	3
phone-1	6	0	3	3
phone-2	6	0	2	3
phone-5	7	0	2	3
phone-6	7	0	1	3
phone-7	7	0	2	3
phone-8	7	0	0	1

Figure 4-5: Minimum number of correct examples required to synthesize a correct program.

With the 1-Delete loss function, the minimum number of required correct input-output examples is always 0 — the implementation synthesizes, for every benchmark problem, a correct program that matches every input-output example in the original clean dataset even when given a dataset in which every output is corrupted. This result highlights how 1) discrete noise sources produce noisy outputs that leave a significant amount of information from the original uncorrupted output available in the corrupted output and 2) a loss function that matches the noise source can enable the synthesis technique to exploit this information to produce correct programs even in the face of substantial noise.

With the DL loss function, the implementation synthesizes a correct program for 8 of the 16 benchmarks when all outputs in the dataset are corrupted. For 7 of the remaining 8 benchmarks the technique requires 2 correct input-output examples to synthesize the correct program. The remaining benchmark requires 3 correct examples. The general pattern is that the technique tends to require correct examples

Benchmark	Data set size	Number of required correct examples		
		1-Delete	DL	0/1
bikes	6	0	0	3
bikes-long	24	0	0	9
bikes-long-repeat	58	0	0	10
bikes-short	6	0	0	3
dr-name	4	0	0	2
dr-name-long	50	0	0	6
dr-name-short	4	0	0	2
firstname	4	0	0	2
firstname-long	54	0	0	9
firstname-long-repeat	204	0	0	30
firstname-short	4	0	0	2
lastname	4	0	1	1
lastname-long	54	0	0	9
lastname-long-repeat	204	0	0	30
lastname-short	4	0	1	1
initials	4	0	2	2
initials-long	54	0	12	12
initials-short	4	0	2	2
reverse-name	6	0	0	2
reverse-name-short	6	0	0	2
name-combine	6	0	0	2
name-combine-2	4	0	0	2
name-combine-2-long	54	0	0	7
name-combine-2-short	4	0	0	2
name-combine-3	6	0	0	2
name-combine-3-long	50	0	0	6
name-combine-3-long-repeat	200	0	0	22
name-combine-3-short	6	0	0	2
phone	6	0	2	3
phone-long	100	0	29	29
phone-long-repeat	400	0	119	119
phone-short	6	0	2	3
phone-1	6	0	3	3
phone-1-long	100	0	28	28
phone-1-long-repeat	400	0	116	116
phone-1-short	6	0	3	3
phone-2	6	0	2	3
phone-2-long	100	0	27	28
phone-2-long-repeat	400	0	110	110
phone-2-short	6	0	2	3
phone-5	7	0	2	3
phone-5-long	100	0	10	24
phone-5-long-repeat	400	0	35	84
phone-5-short	7	0	2	3
phone-6	7	0	1	3
phone-6-long	100	0	11	27
phone-6-long-repeat	400	0	31	107
phone-6-short	7	0	1	3
phone-7	7	0	2	3
phone-7-long	100	0	7	25
phone-7-long-repeat	400	0	21	102
phone-7-short	7	0	2	3
phone-8	7	0	0	1
phone-8-long	100	0	0	3
phone-8-long-repeat	400	0	0	9
phone-8-short	7	0	0	1

Figure 4-6: Minimum number of correct examples required to synthesize a correct program.

when the output strings are short. The synthesized incorrect programs typically use less of the input string.

These results highlight how the DL loss function still extracts significant useful information available in outputs corrupted with discrete noise sources. But in comparison with the 1-Delete loss function, the DL loss function is not as good a match for the character deletion noise source. The result is that the synthesis technique, when working with the DL loss function, works better with longer inputs, some-

times encounters incorrect programs that fit the corrupted data better, and therefore sometimes requires correct inputs to synthesize the correct program.

The performance advantage of the 1-Delete loss function in this scenario can be explained using a concept of an optimal loss function. The concept of an optimal loss function formalizes this match between the loss function and the noise source. I explore the concept of an optimal loss function in Chapter 5.

With the 0/1 loss function, the technique always requires at least one and usually more correct inputs to synthesize the correct program. In contrast to the 1-Delete and DL loss functions, the 0/1 loss function does not extract information from corrupted outputs. To synthesize a correct program with the 0/1 loss function in this scenario, the technique must effectively ignore the corrupted outputs to synthesize the program working only with information from the correct outputs. It therefore always requires at least one and usually more correct outputs before it can synthesize the correct program.

I present the results for all SyGuS 2018 benchmarks (except benchmarks on which our implementation timed out on) in Figure 4-6.

4.3 Noisy Data Sets, Character Replacements

I next present results for my implementation running on larger data sets with character replacements. The phone-*-long-repeat benchmarks within the SyGuS 2018 benchmarks contain transformations over phone numbers. The datasets for these benchmarks contain 400 input-output examples, including repeated input-output examples.

For each of these phone-*-long-repeat benchmark problems on which my technique terminates with bounded scope height threshold 4 (Section 4.1), I construct a noisy dataset as follows. For each digit in each output string in the dataset, I flip a biased coin. With probability b, I replace the digit with a uniformly chosen random digit (so that each digit in the noisy output is not equal to the original digit in the uncorrupted output with probability $9/10 \times b$).

I then run my implementation on each benchmark problem with the noisy dataset

using the tradeoff objective function $U_\lambda(l, r, c) = l + \lambda \times r + \gamma \times c$ with complexity measure $c = \text{Size}(p)$, uniform regularizer, and the following n-Substitution loss function:

Definition 11. n-Substitution Loss Function: *The n-Substitution loss function $\mathcal{L}_{nS}(\boldsymbol{x}, \boldsymbol{y})$, for each example, counts the number of positions where the noisy output does not agree with the output from the synthesized program. If the synthesized program produces an output that is longer or shorter than the output in the noisy dataset, the loss function is ∞:*

$$\mathcal{L}_{nS}(\langle z_1, \ldots z_n \rangle, \langle y_1, \ldots y_n \rangle) = \sum_{i=1}^{n} L_{nS}(z_i, y_i), \text{ where}$$

$$L_{nS}(z, y) = \begin{cases} \infty & |z| \neq |y| \\ \sum_{i=1}^{|z|} 1 \text{ if } z[i] \neq y[i] \text{ else } 0 & |z| = |y| \end{cases}$$

I run the implementation for all combinations of the bounded scope threshold $b \in \{0.2, 0.4, 0.6\}$ and $\lambda \in \{0.001, 0.1\}$. For every combination of b and λ, and for every one of the phone-*-long-repeat benchmarks in the SyGuS 2018 benchmark set, the implementation synthesizes a correct program that produces the same outputs as in the original (hidden) clean dataset.

These results highlight, once again, the ability of my technique to work with loss functions that match the characteristics of discrete noise sources to synthesize correct programs even in the face of substantial noise.

4.4 Approximate Program Synthesis

For the benchmarks in Figure 4-7, a correct program does not exist within the DSL at bounded scope threshold 2. Figure 4-7 presents results from my implementation on the clean (noise-free) benchmark datasets with the DL loss function, $\mathsf{Size}(p)$ complexity measure, lexicographic objective function U_L, uniform regularizer $\mathcal{R}_U$, and bounded scope threshold 2. The first column presents the name of the benchmark. The next four columns present the number of input-output examples in the benchmark dataset, the DL loss incurred by the synthesized program over the entire dataset,

the sum of the lengths of the output strings of the dataset (the DL loss for an empty output would be this sum), and the size of the synthesized program.

Benchmark	Data set size	DL loss	Output size	Program size
name-combine-4	5	10	49	16
phone-3	7	14	91	11
phone-4	6	6	66	17
phone-9	7	14	99	21
phone-10	7	14	120	21

Figure 4-7: Approximate program synthesis with DL loss function.

For the phone-* benchmarks, a correct program outputs the entire input telephone number but changes the punctutation, for example by including an area code in parentheses. The synthesized approximate programs correctly preserve the telephone number but apply only some of the punctuation changes. The result is $2 = 14/7$ characters incorrect per output for all but phone-4, which has 1 character per output incorrect. Each output is between $13 = 91/7$ and $17 = 120/7$ characters long. For name-combine-4, the synthesized approximate program correctly extracts the last name, inserts a comma and a period, but does not extract the initial of the first name. These results highlight the ability of my technique to approximate a correct program when the correct program does not exist in the program search space.

Figures 4-8, 4-9, 4-10, and 4-11 present results from my implementation on the clean (noise-free) benchmark datasets in SyGuS 2018. The first column presents the name of the benchmark. The next column presents the number of input-output examples in the given benchmark. The next column presents the sum of the lengths of the output strings of the dataset. The next two columns present results for the technique running with bounded scope height threshold $d = 1$ and 2, or 3 and 4, respectively. Each column has four subcolumns: the first presents the running time on that benchmark problem (in seconds). The second presents the number of states in the FTA (in thousands of states). The third presents the DL loss of the synthesized program over the entire dataset (compare this DL loss with the sum of the output lengths over the dataset). The fourth presents the size of the synthesized program.

Threshold			1				2			
Benchmark name	n	out size	time(sec)	FTA size	loss	size	time(sec)	FTA size	loss	size
bikes	6	33	0.18	14.28	0	8	1.04	191.6	0	8
bikes-long	24	136	0.37	135.42	0	8	2.85	1695.02	0	8
bikes-long-repeat	58	325	0.68	135.46	0	8	5.81	1695.06	0	8
bikes-short	6	33	0.13	14.28	0	8	1.11	191.6	0	8
dr-name	4	36	0.49	63.76	4	11	8.66	1693.15	0	14
dr-name-long	50	515	1.24	356.55	50	11	22.5	9844.45	0	14
dr-name-long-repeat	150	1545	2.34	3565.15	150	11	59.33	98444.15	0	14
dr-name-short	4	36	0.55	63.76	4	11	8.53	1693.15	0	14
firstname	4	20	0.26	15.95	0	8	1.67	133.16	0	8
firstname-long	54	335	1.47	161.35	0	8	16.14	1333.45	0	8
firstname-long-repeat	204	1280	3.64	1613.2	0	8	47.74	13334.2	0	8
firstname-short	4	20	0.23	15.95	0	8	1.61	133.16	0	8
initials	4	16	0.23	15.97	8	6	1.64	168.58	4	12
initials-long	54	216	1.23	143.25	108	6	14.83	1669.35	54	12
initials-long-repeat	204	816	3.14	1432.2	408	6	48.23	16693.2	204	12
initials-short	4	16	0.22	15.97	8	6	1.61	168.58	4	12
lastname	4	30	0.42	38.48	0	10	4.45	591.56	0	10
lastname-long	54	356	1.48	186.05	0	10	18.5	2316.35	0	10
lastname-long-repeat	204	1334	3.69	1860.2	0	10	60.44	23163.2	0	10
lastname-short	4	30	0.44	38.48	0	10	4.25	591.56	0	10
name-combine	6	81	0.44	69.97	6	8	9.55	3263.99	0	21
name-combine-long	50	691	1.53	546.75	50	8	41.16	20330.85	0	21
name-combine-long-repeat	204	2818	8.85	9717.2	204	8	464.71	392615.2	0	21
name-combine-short	6	81	0.45	69.97	6	8	10.23	3263.99	0	21
name-combine-2	4	32	0.5	49.25	4	11	6.43	1124.6	4	11
name-combine-2-long	54	497	2.03	407.45	54	11	47.53	9703.35	54	11
name-combine-2-long-repeat	204	1892	5.35	4074.2	204	11	161.69	97033.2	204	11
name-combine-2-short	4	32	0.51	49.25	4	11	6.5	1124.6	4	11
name-combine-3	6	56	0.33	38.94	12	13	4.12	984.52	6	16
name-combine-3-long	50	476	1.2	288.15	100	13	18.06	6511.15	50	16
name-combine-3-long-repeat	200	1904	2.53	2881.2	400	13	59.13	65111.2	200	16
name-combine-3-short	6	56	0.34	38.94	12	13	3.98	984.52	6	16
name-combine-4	5	49	0.34	52.26	15	13	5.1	1679.88	10	16
name-combine-4-long	50	526	1.39	362.65	150	13	22.88	9825.05	100	16
name-combine-4-long-repeat	200	2104	3.15	3626.2	600	13	77.55	98250.2	400	16
name-combine-4-short	5	49	0.36	52.26	15	13	4.92	1679.88	10	16
phone	6	18	0.13	7.35	0	6	0.55	48.79	0	6
phone-long	100	300	0.71	734.1	0	6	4.06	4878.1	0	6
phone-long-repeat	400	1200	1.51	734.4	0	6	13.93	4878.4	0	6
phone-short	6	18	0.1	7.35	0	6	0.55	48.79	0	6
phone-1	6	18	0.13	7.35	0	6	0.57	48.79	0	6
phone-1-long	100	300	0.76	734.1	0	6	4.22	4878.1	0	6
phone-1-long-repeat	400	1200	1.52	734.4	0	6	14.18	4878.4	0	6
phone-1-short	6	18	0.1	7.35	0	6	0.55	48.79	0	6
phone-2	6	18	0.12	7.35	0	8	0.57	48.79	0	8
phone-2-long	100	300	0.69	734.1	0	8	4.09	4878.1	0	8
phone-2-long-repeat	400	1200	1.55	734.4	0	8	17.56	4878.4	0	8
phone-2-short	6	18	0.12	7.35	0	8	0.56	48.79	0	8
phone-3	7	91	0.39	42.06	14	11	5.13	1826.9	14	11
phone-3-long	100	1300	1.54	4205.1	200	11	51.3	182689.1	200	11
phone-3-long-repeat	400	5200	4.28	4205.4	800	11	182.03	182689.4	800	11
phone-3-short	7	91	0.35	42.06	14	11	5.03	1826.9	14	11
phone-4	6	66	0.26	39.66	12	8	4.11	1498.16	6	17
phone-4-long	100	1100	1.44	3965.1	200	8	43.97	149815.1	100	17
phone-4-long-repeat	400	4400	3.85	3965.4	800	8	152.39	149815.4	400	17
phone-4-short	6	66	0.29	39.66	12	8	4.17	1498.16	6	17
phone-5	7	15	0.15	5.57	0	8	0.64	5.57	0	8
phone-5-long	100	240	1.05	556.1	0	8	4.77	556.1	0	8
phone-5-long-repeat	400	960	2.58	556.4	0	8	14.38	556.4	0	8
phone-5-short	7	15	0.15	5.57	0	8	0.63	5.57	0	8
phone-6	7	21	0.25	12.85	0	10	1.39	72.62	0	10
phone-6-long	100	300	1.44	1284.1	0	10	14.09	7261.1	0	10
phone-6-long-repeat	400	1200	3.98	1284.4	0	10	42.03	7261.4	0	10
phone-6-short	7	21	0.3	12.85	0	10	1.39	72.62	0	10

Figure 4-8: Runtimes, FTA sizes, synthesized program's loss, and synthesized program's size for SyGuS 2018 benchmarks under DL loss function.

An entry - indicates that the implementation did not terminate.

4.5 Discussion

Practical Applicabilty: The experimental results show that my technique is effective at solving string manipulation program synthesis problems with modestly sized solutions like those present in the SyGuS 2018 benchmarks. More specifically, the

Threshold			1				2			
Benchmark name	n	out size	time(sec)	FTA size	loss	size	time(sec)	FTA size	loss	size
phone-7	7	21	0.22	12.85	0	10	1.34	72.62	0	10
phone-7-long	100	300	1.5	1284.1	0	10	12.99	7261.1	0	10
phone-7-long-repeat	400	1200	4.25	1284.4	0	10	44.31	7261.4	0	10
phone-7-short	7	21	0.22	12.85	0	10	1.41	72.62	0	10
phone-8	7	21	0.22	12.85	0	10	1.33	72.62	0	10
phone-8-long	100	300	1.45	1284.1	0	10	14.08	7261.1	0	10
phone-8-long-repeat	400	1200	3.96	1284.4	0	10	46.61	7261.4	0	10
phone-8-short	7	21	0.22	12.85	0	10	1.4	72.62	0	10
phone-9	7	99	0.86	163.06	21	8	43.75	12299.47	14	21
phone-9-long	100	1440	5.19	16305.1	300	8	555.42	1229985.1	200	21
phone-9-long-repeat	400	5760	17.01	16305.4	1200	8	-	-	-	-
phone-9-short	7	99	0.9	163.06	21	8	41.61	12299.47	14	21
phone-10	7	120	1.18	187.59	21	8	65.53	18600.06	14	21
phone-10-long	100	1740	6.82	18758.1	300	8	-	-	-	-
phone-10-long-repeat	400	6960	21.71	18758.4	1200	8	-	-	-	-
phone-10-short	7	120	0.98	187.59	21	8	65.15	18600.06	14	21
reverse-name	6	81	0.44	69.97	6	18	9.12	3263.99	0	21
reverse-name-long	50	691	1.45	546.75	50	18	43.63	20330.85	0	21
reverse-name-long-repeat	200	2764	4.07	5467.2	200	18	150.97	203308.2	0	21
reverse-name-short	6	81	0.55	69.97	6	18	10.1	3263.99	0	21
univ-1	6	258	147.5	11954.27	12	8	-	-	-	-
univ-1-long	20	699	98.37	22386.12	40	8	-	-	-	-
univ-1-long-repeat	30	1000	132.04	22386.13	60	8	-	-	-	-
univ-1-short	6	258	170.89	11954.27	12	8	-	-	-	-
univ-2	6	243	99.43	4954.21	17	18	-	-	-	-
univ-2-long	20	744	115.13	27793.82	65	18	-	-	-	-
univ-2-long-repeat	30	1075	145.75	27793.83	98	18	-	-	-	-
univ-2-short	6	243	106.17	4954.21	17	18	-	-	-	-
univ-3	6	122	22.53	1134.63	5	20	-	-	-	-
univ-3-long	20	378	30.38	4930.22	25	20	-	-	-	-
univ-3-long-repeat	30	570	37.74	4930.23	45	20	-	-	-	-
univ-3-short	6	122	25.99	1134.63	5	20	-	-	-	-
univ-4	8	150	22.75	842.34	18	20	-	-	-	-
univ-4-long	20	366	27.1	4510.42	39	20	-	-	-	-
univ-4-long-repeat	30	552	34.4	4510.43	63	20	-	-	-	-
univ-4-short	8	150	25.79	842.34	18	20	-	-	-	-
univ-5	8	150	25.14	1005.5	18	20	-	-	-	-
univ-5-long	20	366	32.39	5708.62	39	20	-	-	-	-
univ-5-long-repeat	30	552	41.96	5708.63	63	20	-	-	-	-
univ-5-short	8	150	31.34	1005.5	18	20	-	-	-	-
univ-6	8	150	38.49	1171.16	18	20	-	-	-	-
univ-6-long	20	366	36.38	6896.62	39	20	-	-	-	-
univ-6-long-repeat	30	552	53.14	6896.63	63	20	-	-	-	-
univ-6-short	8	150	35.31	1171.16	18	20	-	-	-	-

Figure 4-9: Runtimes, FTA sizes, synthesized program's loss, and synthesized program's size for SyGuS 2018 benchmarks under DL loss function.

results highlight how the combination of structure from the DSL, a discrete noise source that preserves some information even in corrupted outputs, and a good match between the loss function and noise source can enable very effective synthesis for data datasets with only a handful of input-output examples even in the presence of substantial noise. Even with as generic a loss function as the 0/1 loss function, the technique is effective at dealing with datasets in which a significant fraction of the outputs are corrupted. I anticipate that these results will generalize to similar classes of program synthesis problems with modestly sized solutions within a tractable and focused class of computations.

I note that my current implementation does not scale to SyGuS 2018 benchmarks with larger solutions. These benchmarks were designed to test the scalability of current and future program synthesis systems. No currently extant program analysis

Threshold			3				4			
Benchmark name	n	out size	time(sec)	FTA size	loss	size	time(sec)	FTA size	loss	size
bikes	6	33	7.0	1532.67	0	8	49.06	6655.88	0	8
bikes-long	24	136	25.45	13114.22	0	8	158.04	57398.12	0	8
bikes-long-repeat	58	325	55.19	13114.26	0	8	364.4	57398.16	0	8
bikes-short	6	33	7.34	1532.67	0	8	54.77	6655.88	0	8
dr-name	4	36	194.82	29685.59	0	14	-	-	-	-
dr-name-long	50	515	-	-	-	-	-	-	-	-
dr-name-long-repeat	150	1545	-	-	-	-	-	-	-	-
dr-name-short	4	36	302.33	29685.59	0	14	-	-	-	-
firstname	4	20	26.0	595.77	0	8	49.09	595.77	0	8
firstname-long	54	335	251.45	5959.55	0	8	547.8	5959.55	0	8
firstname-long-repeat	204	1280	-	-	-	-	-	-	-	-
firstname-short	4	20	27.41	595.77	0	8	45.07	595.77	0	8
initials	4	16	31.26	1450.97	0	22	117.93	5913.66	0	22
initials-long	54	216	370.59	14493.25	0	22	-	-	-	-
initials-long-repeat	204	816	-	-	-	-	-	-	-	-
initials-short	4	16	19.0	1450.97	0	22	121.17	5913.66	0	22
lastname	4	30	59.1	5717.2	0	10	432.69	34502.07	0	10
lastname-long	54	356	249.64	19128.05	0	10	-	-	-	-
lastname-long-repeat	204	1334	-	-	-	-	-	-	-	-
lastname-short	4	30	57.87	5717.2	0	10	481.84	34502.07	0	10
name-combine	6	81	381.69	102428.84	0	21	-	-	-	-
name-combine-long	50	691	-	-	-	-	-	-	-	-
name-combine-long-repeat	204	2818	-	-	-	-	-	-	-	-
name-combine-short	6	81	413.17	102428.84	0	21	-	-	-	-
name-combine-2	4	32	113.3	17414.8	0	24	-	-	-	-
name-combine-2-long	54	497	-	-	-	-	-	-	-	-
name-combine-2-long-repeat	204	1892	-	-	-	-	-	-	-	-
name-combine-2-short	4	32	111.78	17414.8	0	24	-	-	-	-
name-combine-3	6	56	78.62	16834.93	0	22	-	-	-	-
name-combine-3-long	50	476	325.26	110276.25	0	22	-	-	-	-
name-combine-3-long-repeat	200	1904	-	-	-	-	-	-	-	-
name-combine-3-short	6	56	74.39	16834.93	0	22	-	-	-	-
name-combine-4	5	49	139.73	36808.22	5	19	-	-	-	-
name-combine-4-long	50	526	533.41	201894.15	50	19	-	-	-	-
name-combine-4-long-repeat	200	2104	-	-	-	-	-	-	-	-
name-combine-4-short	5	49	139.69	36808.22	5	19	-	-	-	-
phone	6	18	2.37	162.2	0	6	6.16	162.2	0	6
phone-long	100	300	26.93	16219.1	0	6	75.58	16219.1	0	6
phone-long-repeat	400	1200	85.42	16219.4	0	6	251.59	16219.4	0	6
phone-short	6	18	2.31	162.2	0	6	5.47	162.2	0	6
phone-1	6	18	2.37	162.2	0	6	5.86	162.2	0	6
phone-1-long	100	300	26.69	16219.1	0	6	76.31	16219.1	0	6
phone-1-long-repeat	400	1200	86.62	16219.4	0	6	260.76	16219.4	0	6
phone-1-short	6	18	2.49	162.2	0	6	6.3	162.2	0	6
phone-2	6	18	2.16	162.2	0	8	6.37	162.2	0	8
phone-2-long	100	300	26.24	16219.1	0	8	69.53	16219.1	0	8
phone-2-long-repeat	400	1200	86.94	16219.4	0	8	242.99	16219.4	0	8
phone-2-short	6	18	2.05	162.2	0	8	5.78	162.2	0	8
phone-3	7	91	202.67	59912.06	7	20	-	-	-	-
phone-3-long	100	1300	-	-	-	-	-	-	-	-
phone-3-long-repeat	400	5200	-	-	-	-	-	-	-	-
phone-3-short	7	91	201.6	59912.06	7	20	-	-	-	-
phone-4	6	66	126.55	42781.51	6	17	-	-	-	-
phone-4-long	100	1100	-	-	-	-	-	-	-	-
phone-4-long-repeat	400	4400	-	-	-	-	-	-	-	-
phone-4-short	6	66	125.07	42781.51	6	17	-	-	-	-
phone-5	7	15	0.67	5.57	0	8	0.65	5.57	0	8
phone-5-long	100	240	5.28	556.1	0	8	4.27	556.1	0	8
phone-5-long-repeat	400	960	16.45	556.4	0	8	14.09	556.4	0	8
phone-5-short	7	15	0.62	5.57	0	8	0.67	5.57	0	8
phone-6	7	21	7.53	315.42	0	10	22.65	315.42	0	10
phone-6-long	100	300	72.39	31541.1	0	10	313.96	31541.1	0	10
phone-6-long-repeat	400	1200	260.06	31541.4	0	10	-	-	-	-
phone-6-short	7	21	7.63	315.42	0	10	24.87	315.42	0	10

Figure 4-10: Runtimes, FTA sizes, synthesized program's loss, and synthesized program's size for SyGuS 2018 benchmarks under DL loss function.

system of which I am aware can solve these larger problems.

To the extent that the SyGuS 2018 bencharks accurately represent the kinds of program synthesis problems that will be encountered in practice, my results provide encouraging evidence that my technique can help program synthesis systems work effectively with noisy datasets. Important future work in this area will more fully

Threshold			3				4			
Benchmark name	n	out size	time(sec)	FTA size	loss	size	time(sec)	FTA size	loss	size
phone-7	7	21	6.87	315.42	0	10	24.32	315.42	0	10
phone-7-long	100	300	72.47	31541.1	0	10	304.19	31541.1	0	10
phone-7-long-repeat	400	1200	268.87	31541.4	0	10	-	-	-	-
phone-7-short	7	21	7.2	315.42	0	10	22.27	315.42	0	10
phone-8	7	21	7.5	315.42	0	10	23.33	315.42	0	10
phone-8-long	100	300	82.64	31541.1	0	10	327.2	31541.1	0	10
phone-8-long-repeat	400	1200	266.52	31541.4	0	10	-	-	-	-
phone-8-short	7	21	6.83	315.42	0	10	25.22	315.42	0	10
phone-9	7	99	-	-	-	-	-	-	-	-
phone-9-long	100	1440	-	-	-	-	-	-	-	-
phone-9-long-repeat	400	5760	-	-	-	-	-	-	-	-
phone-9-short	7	99	-	-	-	-	-	-	-	-
phone-10	7	120	-	-	-	-	-	-	-	-
phone-10-long	100	1740	-	-	-	-	-	-	-	-
phone-10-long-repeat	400	6960	-	-	-	-	-	-	-	-
phone-10-short	7	120	-	-	-	-	-	-	-	-
reverse-name	6	81	-	-	-	-	-	-	-	-
reverse-name-long	50	691	-	-	-	-	-	-	-	-
reverse-name-long-repeat	200	2764	-	-	-	-	-	-	-	-
reverse-name-short	6	81	370.83	102428.84	0	21	-	-	-	-
univ-1	6	258	-	-	-	-	-	-	-	-
univ-1-long	20	699	-	-	-	-	-	-	-	-
univ-1-long-repeat	30	1000	-	-	-	-	-	-	-	-
univ-1-short	6	258	-	-	-	-	-	-	-	-
univ-2	6	243	-	-	-	-	-	-	-	-
univ-2-long	20	744	-	-	-	-	-	-	-	-
univ-2-long-repeat	30	1075	-	-	-	-	-	-	-	-
univ-2-short	6	243	-	-	-	-	-	-	-	-
univ-3	6	122	-	-	-	-	-	-	-	-
univ-3-long	20	378	-	-	-	-	-	-	-	-
univ-3-long-repeat	30	570	-	-	-	-	-	-	-	-
univ-3-short	6	122	-	-	-	-	-	-	-	-
univ-4	8	150	-	-	-	-	-	-	-	-
univ-4-long	20	366	-	-	-	-	-	-	-	-
univ-4-long-repeat	30	552	-	-	-	-	-	-	-	-
univ-4-short	8	150	-	-	-	-	-	-	-	-
univ-5	8	150	-	-	-	-	-	-	-	-
univ-5-long	20	366	-	-	-	-	-	-	-	-
univ-5-long-repeat	30	552	-	-	-	-	-	-	-	-
univ-5-short	8	150	-	-	-	-	-	-	-	-
univ-6	8	150	-	-	-	-	-	-	-	-
univ-6-long	20	366	-	-	-	-	-	-	-	-
univ-6-long-repeat	30	552	-	-	-	-	-	-	-	-
univ-6-short	8	150	-	-	-	-	-	-	-	-

Figure 4-11: Runtimes, FTA sizes, synthesized program's loss, and synthesized program's size for SyGuS 2018 benchmarks under DL loss function.

investigate interactions between the DSL, the noise source, the loss function, the classes of synthesis problems that occur in practice, and the scalability of the synthesis technique. A full evaluation of the immediate practical applicability of program synthesis for noisy datasets, as well as a meaningful evaluation of program synthesis more generally, awaits this future work.

Noise Sources With Different Characteristics: My experiments largely consider discrete noise sources that preserve some information in corrupted outputs. The results highlight how loss functions like the 1-Delete, DL, and n-Substitution loss functions can enable my technique to extract and exploit this preserved information to enhance the effectiveness of the synthesis. The question may arise how well may my technique perform with noise sources that leave little or even no information intact in corrupted outputs? Here the results from the 0/1 loss function, which does not

aspire to extract any information from corrupted inputs, may be relevant — if the corrupted outputs considered together do not conform to a target computation in the DSL, the technique will, in effect, ignore these corrupted outputs to synthesize the program based on any remaining uncorrupted outputs. A final possibility is that the noise source may systematically produce outputs characteristic of a valid but incorrect computation. Here I would expect the algorithm to require a balance of correct outputs before it would be able to synthesize the correct program.

Chapter 5

Optimal Loss Function and Convergence Properties

I first define the concept of correctness for a synthesis algorithm. Given a noisy dataset $(\boldsymbol{x}, \boldsymbol{y})$ generated by the hidden program p_h, a synthesis algorithm is correct, if it synthesizes a program p, such that, both p and p_h return the same outputs on inputs $\boldsymbol{x}$. This is a standard correctness criteria used in noise-free programming-by-example synthesis [42, 30].

Using this concept of correctness, I define the concept of optimal loss function. Given a noisy dataset and prior information about a noise source (either as an exact probability distribution which introduced noise or as a prior distribution over potential noise sources), the optimal loss function is the loss function that has the highest probability of causing the synthesis algorithm to synthesize a correct program. Using concepts from Bayesian inference, I provide closed form expressions for the optimal loss function given perfect or imperfect information about the noise source and program source.

Given a program p, a dataset size n, an input source ρ_i, and a noise source ρ_N, a random dataset $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$ of size n can be generated using the program p by first randomly sampling n inputs $\boldsymbol{x}$ from input source ρ_i, computing the correct outputs $\boldsymbol{z} = p[\boldsymbol{x}]$, and then randomly sampling the corrupted outputs $\boldsymbol{y}$ from distribution $\rho_N(\cdot \mid \boldsymbol{z})$. Using this concept of a random dataset, I define the concept of convergence

guarantees. Given a finite space of programs, a program source ρ_p, input source ρ_i, and a noise source ρ_N, a synthesis algorithm parameterized by a loss function, regularizer, and a complexity measure, guarantees convergence, if for all $0 \leq \delta < 1$ and all hidden programs p_h, there exists a dataset size n, such that, given a random dataset of size greater than n generated by the hidden program p_h, the synthesis algorithm will synthesize a program equivalent to p_h with probability greater than δ. A program p is equivalent to program p_h if for all inputs p and p_h produce the same outputs. Increasing the dataset size can only improve the probability of synthesizing a program equivalent to the hidden program, when using a synthesis algorithm which guarantees convergence. I formalize conditions on the loss function, input source, and the noise source, which allows the synthesis algorithm to guarantee convergence.

I then apply these concepts to a general class of noise sources and loss functions for text processing program synthesis. These concepts make it possible to prove optimality and convergence relationships that hold between these noise sources and loss functions.

I analyze synthesis algorithms which use the **lexicographic objective function** (Definition 8), i.e., given a loss function $\mathcal{L}$, regularizer $\mathcal{R}$, complexity measure C, and a noisy dataset $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{z})$, a synthesis algorithm synthesizes a program p^*, where:

$$\boldsymbol{z}^* = p^*[\boldsymbol{x}] = \arg\min_{\boldsymbol{z} \in Z^{|\boldsymbol{x}|}} \mathcal{L}(\boldsymbol{z}, \boldsymbol{y}) + \mathcal{R}(\boldsymbol{z}, \boldsymbol{x})$$

$$p^* = \arg\min_{p \in G_{\boldsymbol{x}, \boldsymbol{z}^*}} C(p)$$

5.1 Optimal Loss Function

I first formalize the probability that a synthesized program p is a correct program (i.e., $p[\boldsymbol{x}] = p_h[\boldsymbol{x}]$, where p_h is the hidden program) given a dataset $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$ and prior information about the noise source and the program source. This allows us to formalize the notion of an optimal loss function and an optimal regularizer. An optimal loss function, in the presence of the optimal regularizer, has the highest probability of synthesizing a correct program, given a noisy dataset. Using this concept,

I formalize a framework to design optimal loss functions and optimal regularizers, given prior information about the noise source and the program source.

5.1.1 Optimal Loss Function, Perfect Information

We first consider the case where we have the specific probability distribution ρ_N that characterizes the noise source. Given dataset $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$, let $Pr[p \mid \boldsymbol{x}, \boldsymbol{y}]$ be the posterior probability that the program p returns the same output (over inputs $\boldsymbol{x}$) as a hidden program which could have generated the dataset $\mathcal{D}$. Formally:

$$Pr[p \mid \boldsymbol{x}, \boldsymbol{y}] = \frac{1}{\rho(\boldsymbol{y} \mid \boldsymbol{x})} \sum_{p_h \in G} \mathbb{1}(p[\boldsymbol{x}] = p_h[\boldsymbol{x}]) \rho_p(p_h) \rho_N(\boldsymbol{y} \mid p_h[\boldsymbol{x}]) = \rho_p(G_{\boldsymbol{x}, p[\boldsymbol{x}]}) \rho_N(\boldsymbol{y} \mid p[\boldsymbol{x}])$$

where $G_{\boldsymbol{x}, p[\boldsymbol{x}]}$ denotes the set of programs in the set G which map inputs $\boldsymbol{x}$ to outputs $p[\boldsymbol{x}]$. The function $\mathbb{1} : \mathbb{B} \to 0, 1$, maps true to 1 and false to 0. The optimal prediction p_i maximizes the posterior probability $Pr[p \mid \boldsymbol{x}, \boldsymbol{y}]$:

$$p_i \in \arg\max_{p \in G} \rho_p(G_{\boldsymbol{x}, p[\boldsymbol{x}]}) \rho_N(\boldsymbol{y} \mid p[\boldsymbol{x}]) = \arg\min_{p \in G} (-\log \rho_N(\boldsymbol{y} \mid p[\boldsymbol{x}])) + (-\log \rho_p(G_{\boldsymbol{x}, p[\boldsymbol{x}]}))$$

Note that, all programs $p \in G_{\boldsymbol{x}, p_i[\boldsymbol{x}]}$ are valid optimal predictions.

The optimal predication p minimizes a function F of the following form:

$$F(\boldsymbol{z}, \boldsymbol{x}, \boldsymbol{y}, G) = \mathcal{L}(\boldsymbol{z}, \boldsymbol{y}) + \mathcal{R}(\boldsymbol{x}, \boldsymbol{z})$$

where $\boldsymbol{z} = p[\boldsymbol{x}]$, for some loss function $\mathcal{L}$ (Subsection 2.2.2), and for some regularizer $\mathcal{R}$ (Subsection 2.2.3).
The optimal prediction p_i maximizes the posterior probability, therefore:

$$p_i \in \arg\min_{p \in G} (-\log \rho_p(G_{\boldsymbol{x}, p[\boldsymbol{x}]})) + (-\log \rho_N(\boldsymbol{y} \mid p[\boldsymbol{x}]))$$

Therefore, given a set of programs G, dataset $\mathcal{D}$, and no other information about the hidden program, the hidden noise source, and the hidden program source, the synthesis algorithm will always return the program which maximizes the posterior probability if the loss function $\mathcal{L}(\boldsymbol{z}, \boldsymbol{y}) = (-\log \rho_N(\boldsymbol{y} \mid \boldsymbol{z}))$ and the regularizer $\mathcal{R}(\boldsymbol{x}, \boldsymbol{z}) =$

$(-\log \rho_p(G_{\boldsymbol{x},\boldsymbol{z}}))$.

Hence, the **optimal loss function** and the **optimal regularizer**, in the presence of perfect information, is the negative log of the probability of output $\boldsymbol{z}$ being corrupted to noisy output $\boldsymbol{y}$ and the negative log of the prior probability a program from set $G_{\boldsymbol{x},\boldsymbol{z}}$ is the hidden program.

Note that, even if we have perfect information about the program source and the noise source, the noise source may make it impossible to synthesize a correct program with probability 1, if it can, with positive probability, corrupt the outputs on two positive probability programs to return the same noisy output.

Theorem 4. *Let G be a set of programs. Let $\boldsymbol{x}$ be a vector of inputs ($\rho_i(\boldsymbol{x}) > 0$). Let $p_1, p_2 \in G$ be two programs, such that, $p_1[\boldsymbol{x}] \neq p_2[\boldsymbol{x}]$. Let $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$ be a noisy dataset, such that, $\rho_N(\boldsymbol{y} \mid p_1[\boldsymbol{x}]) > 0$ and $\rho_N(\boldsymbol{y} \mid p_2[\boldsymbol{x}]) > 0$, i.e., both outputs $p_1[\boldsymbol{x}]$ and $p_2[\boldsymbol{x}]$ can be corrupted by the noise source to generate the same noisy output $\boldsymbol{y}$. Then no algorithm can always synthesize a program which returns the hidden program's output on inputs $\boldsymbol{x}$.*

Proof. Both program p_1 and p_2 could have generated the dataset $\mathcal{D}$. Without any additional information about the hidden process, both p_1 and p_2 are valid candidates to be the hidden program, when given dataset $\mathcal{D}$. Therefore, only given dataset $\mathcal{D}$, no algorithm can always synthesize a program which returns the hidden program's output on the given inputs, if there exists even one positive probability program whose outputs on the given inputs are not equal to the outputs of the hidden program and the noise source can corrupt these outputs to the noisy outputs of the given dataset. □

5.1.2 Optimal Loss Function, Imperfect Information

Now consider a scenario where we are presented with imperfect information about the noise source and the program source, i.e., all we know is that the noise source ρ_N corrupting the correct output was selected from a set of noise sources $\mathcal{N}$ with probability $\rho_{\mathcal{N}}(\rho_N)$ and the program source ρ_P was selected from a set of noise sources $\mathcal{P}$ with probability $\rho_{\mathcal{P}}(\rho_P)$.

Given dataset $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$, I assume it was constructed by the following underlying process:

- A program source ρ_p is sampled from the set $\mathcal{P}$ with probability $\rho_{\mathcal{P}}(\rho_p)$ (prior distribution over program source).
- A noise source ρ_N is sampled from the set $\mathcal{N}$ with probability $\rho_{\mathcal{N}}(\rho_N)$ (prior distribution over noise source).
- A hidden program p_h is sampled from the set G with probability $\rho_p(p_h)$.
- n inputs $\boldsymbol{x} = \langle x_1, \ldots x_n \rangle$ are sampled from probability distribution ρ_i with probability $\rho_i(\boldsymbol{x} \mid n)$.
- The sampled noise source ρ_N introduces noise by corrupting outputs $p_h[\boldsymbol{x}]$ to $\boldsymbol{y}$ with probability $\rho_N(\boldsymbol{y} \mid p_h[\boldsymbol{x}])$.

The posterior probability of a program p returning the hidden program's output is:

$$Pr[p \mid \boldsymbol{x}, \boldsymbol{y}] \propto \sum_{\rho_N \in \mathcal{N}} \sum_{\rho_p \in \mathcal{P}} \sum_{p_h \in G} \mathbb{1}(p[\boldsymbol{x}] = p_h[\boldsymbol{x}]) \rho_p(p_h) \rho_N(\boldsymbol{y} \mid p_h[\boldsymbol{x}]) \rho_{\mathcal{N}}(\rho_N) \rho_{\mathcal{P}}(\rho_p)$$

$$= E[\rho_p(G_{\boldsymbol{x}, p[\boldsymbol{x}]})] E[\rho_N(\boldsymbol{y} \mid p[\boldsymbol{x}])]$$

The optimal prediction p_i maximizes the posterior probability, therefore:

$$p_i \in \arg\min_{p \in G}(-\log E[\rho_p(G_{\boldsymbol{x}, p[\boldsymbol{x}]})]) + (-\log E[\rho_N(\boldsymbol{y} \mid p[\boldsymbol{x}])])$$

Therefore, given a set of programs G, dataset $\mathcal{D}$, prior probability distribution over noise sources $\rho_{\mathcal{N}}$, prior probability distribution over program sources $\rho_{\mathcal{P}}$, and no other information about the hidden program, the hidden noise source, and the hidden program source, the synthesis algorithm will always return the program which maximizes the posterior probability if the loss function $\mathcal{L}(\boldsymbol{z}, \boldsymbol{y}) = (-\log E[\rho_N(\boldsymbol{y} \mid \boldsymbol{z})])$ and the regularizer $\mathcal{R}(\boldsymbol{x}, \boldsymbol{z}) = (-\log E[\rho_p(G_{\boldsymbol{x}, \boldsymbol{z}})])$.

Hence, the optimal loss function and the optimal regularizer, in presence of imperfect information, is the negative log of the expected probability of output $\boldsymbol{z}$ being corrupted to noisy output $\boldsymbol{y}$ and the negative log of the expected prior probability a program from set $G_{\boldsymbol{x},\boldsymbol{z}}$ is the hidden program.

5.2 Convergence

I next explore the conditions under which the synthesis algorithm will have convergence guarantees, i.e., with high probability, the synthesis algorithm will synthesize a program p, such that, p and the hidden program p_h have the same outputs on all inputs x, given a finite program space and a large enough dataset. If the program space contains possibly infinite programs, the synthesis algorithm will have convergence guarantees, i.e., with high probability, the synthesis algorithm will synthesize a program p, such that, p and the hidden program p_h have the same outputs on any finite size input vector $\boldsymbol{x}^*$ ($p[\boldsymbol{x}^*] = p_h[\boldsymbol{x}^*]$), given a large enough random dataset.

Definition 12. $\approx$-closed *A set $A \subseteq G$ is $\approx$-closed if for all programs $p \in A$, any program equivalent to p is also in A, i.e., $\forall p \in A. \forall p' \in G.\ p \approx p' \implies p' \in A$.*

Definition 13. separable *A $\approx$-closed set $A \subseteq G$ is separable if there exists a finite set of inputs $\boldsymbol{x}$, such that, for all programs $p \in A$ and $p' \in G - A$, $p[\boldsymbol{x}] \neq p'[\boldsymbol{x}]$, i.e., there exists a finite set of inputs which allows us to infer that a program is within A or within $G - A$.*

Note that the following statement is true:

- If $A \subseteq G$ is separable, then $G - A$ is separable.
- If G is finite, then for all programs $p \in G$, set G_p is separable, i.e., the set of programs equivalent to p is separable.
- Given a vector of n inputs $\boldsymbol{x}$ and n noise-free outputs $\boldsymbol{z}$, the set $G_{\boldsymbol{x},\boldsymbol{z}}$ is separable.

I use the notation $\mathcal{S}$ to denote a synthesis algorithm. Given a dataset $\mathcal{D}$ and a set of programs G, a synthesis algorithm returns a program $p \in G$ ($p = \mathcal{S}(G, \mathcal{D})$).

Given a set of programs G, an input source ρ_i, a program source ρ_p, a noise source ρ_N, a synthesis algorithm $\mathcal{S}$, a dataset size $n > 0$, a positive probability hidden program $p_h \in G$ (i.e.,$\rho_p(p_h) > 0$), and a separable set $A \subseteq G$ containing p_h (i.e.,$p_h \in A$), $Pr[p_s \in A \mid p_h, n]$ is the probability that the synthesis algorithm synthesizes a program in set A, given a random dataset $(\boldsymbol{x}, \boldsymbol{y})$ of size n, constructed with p_h being the hidden program. Formally, $Pr[p_s \in A \mid p_h, n]$ is the probability that the following process returns true:

- Sample n inputs $\boldsymbol{x}$ with probability $\rho_i(\boldsymbol{x} \mid n)$.
- Sample noisy outputs $\boldsymbol{y}$ with probability $\rho_N(\boldsymbol{y} \mid p_h[\boldsymbol{x}])$.
- Return true, if given the noisy dataset, $\mathcal{S}$ synthesizes a program from set A, i.e., $\mathcal{S}(G, (\boldsymbol{x}, \boldsymbol{y})) \in A$.

$Pr[p_s \in A \mid p_h, n]$ measures the probability that a program within the set A (which contains all equivalent programs to p_h and potentially other programs) is synthesized by the algorithm $\mathcal{S}$, given a random noisy dataset of size n generated by the hidden program p_h.

Definition 14. Convergence: *Given a set of programs G, an input source ρ_i, a program source ρ_p, a noise source ρ_N, and a synthesis algorithm $\mathcal{S}$, the synthesis algorithm guaranties convergence if for all positive probability programs $p_h \in G$, for all separable sets $A \subseteq G$ containing p_h (i.e., $p_h \in G$), for all $\delta > 0$, there exists a natural number k, such that for all $n \geq k$:*

$$Pr[p_s \in A \mid p_h, n] \geq (1 - \delta)$$

i.e., for all positive probability hidden programs, for all separable sets A containing p_h, and for all $\delta > 0$, we can find a minimum dataset size k, such that, the probability that the algorithm will synthesize a program in A on a random dataset of size $\geq k$ is $\geq (1 - \delta)$.

Given a set of programs G, an input source ρ_i, a program source ρ_p, a noise source ρ_N, and a synthesis algorithm $\mathcal{S}$, the following statements are true:

- For any input vector $\boldsymbol{x}^*$ and a positive probability hidden program $p_h \in G$, the set $G_{\boldsymbol{x}^*,p_h[\boldsymbol{x}^*]}$ is separable. The synthesis algorithm $\mathcal{S}$ guarantees convergence, if and only if, for all $\delta > 0$, for all positive probability hidden programs p_h, for all input vectors $\boldsymbol{x}^*$, there exists a minimum dataset size k, such that, for all $n \geq k$:

$$Pr[p_s \in G_{\boldsymbol{x}^*,p_h[\boldsymbol{x}^*]} \mid p_h, n] \geq (1 - \delta)$$

i.e., the synthesis algorithm will synthesize a program p which returns the hidden program's outputs on the input vector $\boldsymbol{x}^*$ (i.e., $p[\boldsymbol{x}^*] = p_h[\boldsymbol{x}^*]$) with probability $\geq (1 - \delta)$.

Theorem 5. *For any input vector $\boldsymbol{x}^*$ and a positive probability hidden program $p_h \in G$, the set $G_{\boldsymbol{x}^*,p_h[\boldsymbol{x}^*]}$ is separable. The synthesis algorithm $\mathcal{S}$ guarantees convergence, if and only if, for all $\delta > 0$, for all positive probability hidden programs p_h, for all input vectors $\boldsymbol{x}^*$, there exists a minimum dataset size k, such that, for all $n \geq k$:*

$$Pr[p_s \in G_{\boldsymbol{x}^*,p_h[\boldsymbol{x}^*]} \mid p_h, n] \geq (1 - \delta)$$

i.e., the synthesis algorithm will synthesize a program p which returns the hidden program's outputs on the input vector $\boldsymbol{x}^*$ (i.e., $p[\boldsymbol{x}^*] = p_h[\boldsymbol{x}^*]$) with probability $\geq (1 - \delta)$.

Proof. The $G_{\boldsymbol{x}^*,p_h[\boldsymbol{x}^*]}$ is a separable set containing p_h. Therefore, by definition of convergence, for all $\delta > 0$, for all positive probability hidden programs p_h, for all input vectors $\boldsymbol{x}^*$, there exists a minimum dataset size k, such that, for all $n \geq k$:

$$Pr[p_s \in G_{\boldsymbol{x}^*,p_h[\boldsymbol{x}^*]} \mid p_h, n] \geq (1 - \delta)$$

Given any separable set A, there exists an input vector $\boldsymbol{x}^A$, such that, $\forall p \in A, p' \in G - A . p[\boldsymbol{x}^A] \neq p'[\boldsymbol{x}^A]$. If A contains the hidden program program p_h,

then $G_{\boldsymbol{x}^A, p_h[\boldsymbol{x}^A]} \subseteq A$. Therefore:

$$Pr[p_s \in A \mid p_h, n] \geq Pr[p_s \in G_{\boldsymbol{x}^A, p_h[\boldsymbol{x}^A]} \mid p_h, n]$$

Therefore, if for all $\delta > 0$, for all positive probability hidden programs p_h, for all input vectors $\boldsymbol{x}^A$, there exists a minimum dataset size k, such that, for all $n \geq k$:

$$Pr[p_s \in G_{\boldsymbol{x}^A, p_h[\boldsymbol{x}^A]} \mid p_h, n] \geq (1 - \delta)$$

then $\mathcal{S}$ guarantees convergence. □

- Given a positive probability hidden program $p_h \in G$. If G_{p_h} is separable and $\mathcal{S}$ guarantees convergence, then for all $\delta > 0$, there exists a minimum dataset size k, such that, for a random dataset of size $\geq k$, the synthesis algorithm will synthesize a program equivalent to the hidden program p_h with probability $\geq (1 - \delta)$.

- If G is finite, then for all programs $p \in G$, G_p is separable. The synthesis algorithm $\mathcal{S}$ guarantees convergence, if and only if, for all $\delta > 0$, for all positive probability hidden programs p_h, there exists a minimum dataset size k, such that, for all $n \geq k$:

$$Pr[p_s \in G_{p_h} \mid p_h, n] \geq (1 - \delta)$$

i.e., the synthesis algorithm will synthesize a program p equivalent to the hidden program p_h with probability $\geq (1 - \delta)$.

Theorem 6. *If G is finite, then for all programs $p \in G$, G_p is separable. The synthesis algorithm $\mathcal{S}$ guarantees convergence, if and only if, for all $\delta > 0$, for all positive probability hidden programs p_h, there exists a minimum dataset size k, such that, for all $n \geq k$:*

$$Pr[p_s \in G_{p_h} \mid p_h, n] \geq (1 - \delta)$$

i.e., the synthesis algorithm will synthesize a program p equivalent to the hidden

program p_h with probability $\geq (1-\delta)$.

Proof. If G is finite, then for all programs $p \in G$, G_p is a separable. Therefore, by definition of convergence, for all $\delta > 0$, for all positive probability hidden programs p_h, there exists a minimum dataset size k, such that, for all $n \geq k$:

$$Pr[p_s \in G_{p_h} \mid p_h, n] \geq (1-\delta)$$

Given any separable set A, if $p \in A$, then $G_p \subseteq A$. Therefore:

$$Pr[p_s \in A \mid p_h, n] \geq Pr[p_s \in G_{p_h} \mid p_h, n]$$

Therefore, if for all $\delta > 0$, for all positive probability hidden programs p_h, there exists a minimum dataset size k, such that, for all $n \geq k$:

$$Pr[p_s \in G_{p_h} \mid p_h, n] \geq (1-\delta)$$

then $\mathcal{S}$ guarantees convergence. □

Given a set of programs G and a synthesis algorithm $\mathcal{S}$, within this thesis, we assume that there exists a loss function $\mathcal{L}$ (Subsection 2.2.2), a regularizer $\mathcal{R}$ (Subsection 2.2.3), and a complexity measure C (Subsection 2.2.4), such that, given any noisy datasets $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$, the synthesis algorithm $\mathcal{S}$ synthesizes a program p_s ($\mathcal{S}(G, \mathcal{D}) = p_s$), if and only if,

$$G_{\boldsymbol{x}, p_s[\boldsymbol{x}]} \subseteq \arg\min_{p \in G} \mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}) + \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}])$$

$$p_s \in \arg\min_{p \in G_{\boldsymbol{x}, p_s[\boldsymbol{x}]}} C(p)$$

I use the notation $(\mathcal{L}, \mathcal{R}, C)$ to denote a synthesis algorithm $\mathcal{S}$ which falls within the framework described above. The analysis, within this thesis, will introduce restrictions on the noise source, loss function and the input source, with minimal restructions on

the program source, regularizer, and the complexity measure. Note that the complexity measure C here, introduces a total ordering on the set of programs G.

5.2.1 Differentiating Input Distributions

Even in the absence of noise, the input source may hinder a synthesis algorithm's ability to infer if the hidden program belongs to a separable set A or not. For example, consider two programs $p_1, p_2 \in G$, and a separable set $A \subseteq G$, such that, $p_1 \in A$ but $p_2 \in G - A$. Consider an input source which only generates vectors $\boldsymbol{x}$, such that, p_1 and p_2 have the same outputs on input $\boldsymbol{x}$ (i.e., $p_1[\boldsymbol{x}] = p_2[\boldsymbol{x}]$). For such an input source, the synthesis algorithm, even in the absence of noise, cannot differentiate between datasets produced assuming p_1 is the underlying program and from the datasets produced assuming p_2 is the underlying program. Therefore, to guarantee convergence, we need to constrain the input source.

Let d be some distance metric which measures the distance between two noise-free outputs $z_1, z_2 \in Z$. I use the notation $d(\boldsymbol{z}, \boldsymbol{z'})$ to denote the sum of distance over individual elements, i.e.,

$$d(\langle z_1, \ldots z_n \rangle, \langle z'_1, \ldots z'_n \rangle) = \sum_{i=1}^{n} d(z_i, z'_i)$$

Definition 15. Differentiating Input Source: *Given a set of programs G and a distance metric d over elements from the output set Z, an input source ρ_i is* **differentiating** *if, for all $\delta > 0$, for all $\epsilon > 0$, all programs $p_h \in G$, for all separable $A \subseteq G$ containing p_h, there exists a minimum dataset size k, such that, for all $n \geq k$, the following process returns true with probability greater than equal to $(1 - \delta)$:*

- *Sample $\boldsymbol{x}$ of size n from the distribution $\rho_i(\boldsymbol{x} \mid n)$.*
- *Return true if $\forall p \in G - A$. $d(p[\boldsymbol{x}], p_h[\boldsymbol{x}]) \geq \epsilon$.*

Formally, given a set of programs G and a distance metric d, an input source ρ_i is **differentiating**, if for all $\delta > 0$, for all $\epsilon > 0$, for all programs p_h, and all separable sets $A \subseteq G$ containing p_h, there exists a natural number k, such that for all natural

numbers $n \geq k$, the following statement is true:

$$\sum_{\boldsymbol{x} \in X^n} \mathbb{1}(\forall p \in G - A.\ d(p_h[\boldsymbol{x}], p[\boldsymbol{x}]) \geq \epsilon) \rho_i(\boldsymbol{x} \mid n) \geq (1 - \delta)$$

5.2.2 Differentiating Noise Sources

Even if we are given an input source which allows us to differentiate between programs in a separable set A and other programs in G in the absence of noise, the noise source can make convergence impossible. For example, consider a noise source which replaces all outputs with the same value. No information about the hidden program's outputs can be inferred from this value. A synthesis algorithm, in this case, cannot infer any information about the hidden program from such a dataset.

Therefore, we have to place constraints over the noise source and the loss function to guarantee convergence.

Definition 16. Differentiating Noise Source: *Given a set of programs G, a distance metric d, and a loss function $\mathcal{L}$, a noise source ρ_N is* **differentiating**, *if for all $\delta > 0$ and $\gamma > 0$, there exists a natural number k, and $\epsilon \in \mathbb{R}^+$, such that for all $n \geq k$, for all vectors $\boldsymbol{z}_h$ of length n, the following is true:*

$$\sum_{\boldsymbol{y} \in Y^n} \mathbb{1}\Big(\forall \boldsymbol{z} \in Z^n.\ \mathcal{L}(\boldsymbol{z}, \boldsymbol{y}) - \mathcal{L}(\boldsymbol{z}_h, \boldsymbol{y}) \leq \gamma \implies d(\boldsymbol{z}, \boldsymbol{z}_h) < \epsilon\Big) \rho_N(\boldsymbol{y} \mid \boldsymbol{z}_h) \geq (1 - \delta)$$

If we are using the optimal loss function for the given noise source (Subsection 5.1.1), the above condition reduces to:

$$\sum_{\boldsymbol{y} \in Y^n} \mathbb{1}\left(\forall \boldsymbol{z} \in Z^n.\ \frac{\rho_N(\boldsymbol{y} \mid \boldsymbol{z}_h)}{\rho_N(\boldsymbol{y} \mid \boldsymbol{z})} \leq \gamma \implies d(\boldsymbol{z}, \boldsymbol{z}_h) < \epsilon\right) \rho_N(\boldsymbol{y} \mid \boldsymbol{z}_h) \geq (1 - \delta)$$

Convergence: Convergence is guaranteed in presence of a differentiating input source and a differentiating noise source.

Theorem 7. *Given a set of programs G, a program source ρ_p, a loss function $\mathcal{L}$, a complexity measure C, a regularizer $\mathcal{R}$ (such that, for all $\boldsymbol{x}$ and $\boldsymbol{z}$, $\rho_p(G_{\boldsymbol{x},\boldsymbol{z}}) > 0 \implies$*

$\mathcal{R}(\boldsymbol{x}, \boldsymbol{z}) < \infty$*), a* **differentiating input source** ρ_i*, and a* **differentiating noise source** ρ_N*, then the synthesis algorithm* $(\mathcal{L}, \mathcal{R}, C)$ *will guarantee convergence.*

Proof. Given a $\delta > 0$, let $\delta_i > 0$ and $\delta_N > 0$ be two real numbers, such that, $\delta = \delta_i + \delta_N$. Let p_h be a positive probability program in G, i.e., $\rho_p(p_h) > 0$. Let $A \subseteq G$ be a separable set containing p_h. Let $\gamma_o = \max_{n \in \mathbb{N}, \boldsymbol{x} \in X^n} \mathcal{R}(\boldsymbol{x}, p_h[\boldsymbol{x}])$ Note that, $\gamma_o < \infty$. Since ρ_N is a differentiating noise source, there exists a dataset size k_N, such that, for all $n \geq k_N$, for all vectors $\boldsymbol{z}_h$ of length n, the following is true:

$$\sum_{\boldsymbol{y} \in Y^n} \mathbb{1}\Big(\forall \boldsymbol{z} \in Z^n.\ \mathcal{L}(\boldsymbol{z}, \boldsymbol{y}) - \mathcal{L}(\boldsymbol{z}_h, \boldsymbol{y}) \leq \gamma_o \implies d(\boldsymbol{z}, \boldsymbol{z}_h) < \epsilon_o\Big) \rho_N(\boldsymbol{y} \mid \boldsymbol{z}_h) \geq (1 - \delta_N)$$

Since ρ_i is a differentiating input source, there exists a dataset size k_i, such that, for all $n \geq k_i$,

$$\sum_{\boldsymbol{x} \in X^n} \mathbb{1}(\forall p \in G - A.\ d(p_h[\boldsymbol{x}], p[\boldsymbol{x}]) \geq \epsilon_o) \rho_i(\boldsymbol{x} \mid n) \geq (1 - \delta_i)$$

Let $n \geq \max(k_N, k_i)$,

$$\begin{aligned} Pr[p_s \in A \mid p_h, n] \geq \sum_{\boldsymbol{x} \in X^n} \sum_{\boldsymbol{y} \in Y^n} \mathbb{1}(\exists p_s \in A.\ p_s \in \operatorname*{arg\,min}_{p \in G_{\boldsymbol{x}, p_s[\boldsymbol{x}]}} C(p) \\ \wedge\ G_{\boldsymbol{x}, p_s[\boldsymbol{x}]} \subseteq \operatorname*{arg\,min}_{p \in G} \mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}) + \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}])) \rho_N(\boldsymbol{y} \mid p_h[\boldsymbol{x}]) \rho_i(\boldsymbol{x} \mid n) \\ \geq \sum_{\boldsymbol{x} \in X^n, \boldsymbol{y} \in Y^n} \mathbb{1}\Big(\forall p \in G - A. \mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}) + \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}]) > \mathcal{L}(p_h[\boldsymbol{x}], \boldsymbol{y}) + \mathcal{R}(\boldsymbol{x}, p_h[\boldsymbol{x}])\Big) \\ \rho_i(\boldsymbol{x} \mid n) \rho_N(\boldsymbol{y} \mid p_h[\boldsymbol{x}]) \end{aligned}$$

Note that $\gamma_o \geq \mathcal{R}(\boldsymbol{x}, p_h[\boldsymbol{x}]) - \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}])$.

$$\geq \sum_{\boldsymbol{x} \in X^n, \boldsymbol{y} \in Y^n} \mathbb{1}\Big(\forall p \in G - A. \mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}) - \mathcal{L}(p_h[\boldsymbol{x}], \boldsymbol{y}) > \gamma_o\Big) \rho_i(\boldsymbol{x} \mid n) \rho_N(\boldsymbol{y} \mid p_h[\boldsymbol{x}])$$

$$\begin{aligned} \geq \sum_{\boldsymbol{x} \in X^n, \boldsymbol{y} \in Y^n} \mathbb{1}\Big(\forall p \in G - A. \mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}) - \mathcal{L}(p_h[\boldsymbol{x}], \boldsymbol{y}) > \gamma_o\Big) \rho_N(\boldsymbol{y} \mid p_h[\boldsymbol{x}]) \\ \mathbb{1}(\forall p \in G - A. d(p_h[\boldsymbol{x}], p[\boldsymbol{x}]) \geq \epsilon_o)\ \rho_i(\boldsymbol{x} \mid n) \end{aligned}$$

$$\geq \sum_{\boldsymbol{x}\in X^n, \boldsymbol{y}\in Y^n} \mathbb{1}\Big(\forall \boldsymbol{z} \in Z^n . d(p_h[\boldsymbol{x}], \boldsymbol{z}) \geq \epsilon_o \implies \mathcal{L}(\boldsymbol{z}, \boldsymbol{y}) - \mathcal{L}(p_h[\boldsymbol{x}], \boldsymbol{y}) > \gamma_o\Big)$$

$$\rho_N(\boldsymbol{y} \mid p_h[\boldsymbol{x}]) \mathbb{1}(\forall p \in G - A . d(p_h[\boldsymbol{x}], p[\boldsymbol{x}]) \geq \epsilon_o)\ \rho_i(\boldsymbol{x} \mid n)$$

$$\geq (1-\delta_N) \sum_{\boldsymbol{x}\in X^n} \mathbb{1}(\forall p \in G - A.\ d(p_h[\boldsymbol{x}], p[\boldsymbol{x}]) \geq \epsilon) \rho_i(\boldsymbol{x} \mid n) \geq (1-\delta_N)(1-\delta_i) \geq (1-\delta)$$

□

Linear Combination of Loss Functions:

Given a set of programs G, a distance metric d, loss functions $\mathcal{L}_a, \mathcal{L}_b$, and a noise source ρ_N which is **differentiating** w.r.t. both $\mathcal{L}_a$, $\mathcal{L}_b$, then ρ_N is **differentiating** w.r.t. any linear combination of $\mathcal{L}_a$ and $\mathcal{L}_b$ with positive coefficients. Formally,

Theorem 8. *Given a set of programs G, a distance metric d, loss functions $\mathcal{L}_a$, $\mathcal{L}_b$, two real numbers $a, b \geq 0$, and a noise source ρ_N. Let $\mathcal{L}$ be a loss function, such that,*

$$\mathcal{L}(\boldsymbol{z}, \boldsymbol{y}) = a\mathcal{L}_a(\boldsymbol{z}, \boldsymbol{y}) + b\mathcal{L}_b(\boldsymbol{z}, \boldsymbol{y})$$

If for all $\delta_a, \delta_b > 0$ and $\gamma_a, \gamma_b > 0$, there exists natural numbers k_a, k_b, and $\epsilon_a, \epsilon_b \in \mathbb{R}^+$, such that, for all vector $\boldsymbol{z}_a$ of length $n_a \geq k_a$, and vector $\boldsymbol{z}_b$ of length $n_b \geq k_b$, the following is true:

$$\sum_{\boldsymbol{y}\in Y^{n_a}} \mathbb{1}\Big(\forall \boldsymbol{z} \in Z^{n_a}.\ \mathcal{L}_a(\boldsymbol{z}, \boldsymbol{y}) - \mathcal{L}_a(\boldsymbol{z}_a, \boldsymbol{y}) \leq \gamma_a \implies d(\boldsymbol{z}, \boldsymbol{z}_a) < \epsilon_a\Big) \rho_N(\boldsymbol{y} \mid \boldsymbol{z}_a) \geq (1-\delta)$$

$$\sum_{\boldsymbol{y}\in Y^{n_b}} \mathbb{1}\Big(\forall \boldsymbol{z} \in Z^{n_b}.\ \mathcal{L}_b(\boldsymbol{z}, \boldsymbol{y}) - \mathcal{L}_b(\boldsymbol{z}_b, \boldsymbol{y}) \leq \gamma_b \implies d(\boldsymbol{z}, \boldsymbol{z}_b) < \epsilon_b\Big) \rho_N(\boldsymbol{y} \mid \boldsymbol{z}_b) \geq (1-\delta_b)$$

then for all $\delta > 0$ and $\gamma > 0$, there exists a natural number k and $\epsilon \in \mathbb{R}^+$, such that, for all vectors $\boldsymbol{z}_h$ of length $n \geq k$, the following is true:

$$\sum_{\boldsymbol{y}\in Y^{n}} \mathbb{1}\Big(\forall \boldsymbol{z} \in Z^{n}.\ \mathcal{L}(\boldsymbol{z}, \boldsymbol{y}) - \mathcal{L}(\boldsymbol{z}_h, \boldsymbol{y}) \leq \gamma \implies d(\boldsymbol{z}, \boldsymbol{z}_h) < \epsilon\Big) \rho_N(\boldsymbol{y} \mid \boldsymbol{z}_h) \geq (1-\delta)$$

Proof. Given a $\delta > 0$ and $\gamma > 0$, let γ_a and γ_b be two real numbers, such that, $\gamma \geq a\gamma_a + b\gamma_b$. Let δ_a, δ_b be two real numbers, such that, $\delta < \delta_a + \delta_b$.

There exists natural numbers k_a, k_b and real numbers $\epsilon_a, \epsilon_b > 0$, such that, for all

vector $\boldsymbol{z}_h$ of length $n \geq \max(k_a, k_b)$, the following is true:

$$\sum_{\boldsymbol{y} \in Y^n} \mathbb{1}\Big(\forall \boldsymbol{z} \in Z^n.\ \mathcal{L}_a(\boldsymbol{z}, \boldsymbol{y}) - \mathcal{L}_a(\boldsymbol{z}_h, \boldsymbol{y}) \leq \gamma_a \implies d(\boldsymbol{z}, \boldsymbol{z}_h) < \epsilon_a\Big) \rho_N(\boldsymbol{y} \mid \boldsymbol{z}_h) \geq (1 - \delta_a)$$

$$\sum_{\boldsymbol{y} \in Y^n} \mathbb{1}\Big(\forall \boldsymbol{z} \in Z^n.\ \mathcal{L}_b(\boldsymbol{z}, \boldsymbol{y}) - \mathcal{L}_b(\boldsymbol{z}_h, \boldsymbol{y}) \leq \gamma_b \implies d(\boldsymbol{z}, \boldsymbol{z}_h) < \epsilon_b\Big) \rho_N(\boldsymbol{y} \mid \boldsymbol{z}_h) \geq (1 - \delta_b)$$

Let ϵ be a real number, such that, $\epsilon \geq \epsilon_a$, and $\epsilon \geq \epsilon_b$.

$$\sum_{\boldsymbol{y} \in Y^n} \mathbb{1}\Big(\forall \boldsymbol{z} \in Z^n.\ \mathcal{L}(\boldsymbol{z}, \boldsymbol{y}) - \mathcal{L}(\boldsymbol{z}_h, \boldsymbol{y}) \leq \gamma \implies d(\boldsymbol{z}, \boldsymbol{z}_h) < \epsilon\Big) \rho_N(\boldsymbol{y} \mid \boldsymbol{z}_h) \geq$$

$$\sum_{\boldsymbol{y} \in Y^n} \mathbb{1}\Big(\forall \boldsymbol{z} \in Z^n.\ \mathcal{L}_a(\boldsymbol{z}, \boldsymbol{y}) - \mathcal{L}_a(\boldsymbol{z}_h, \boldsymbol{y}) \leq \gamma_a \implies d(\boldsymbol{z}, \boldsymbol{z}_h) < \epsilon_a\Big)$$

$$\mathbb{1}\Big(\forall \boldsymbol{z} \in Z^n.\ \mathcal{L}_b(\boldsymbol{z}, \boldsymbol{y}) - \mathcal{L}_b(\boldsymbol{z}_h, \boldsymbol{y}) \leq \gamma_b \implies d(\boldsymbol{z}, \boldsymbol{z}_h) < \epsilon_b\Big) \rho_N(\boldsymbol{y} \mid \boldsymbol{z}_h) \geq$$

$$\max\Big(\frac{1}{2}, \sum_{\boldsymbol{y} \in Y^n} \mathbb{1}\Big(\forall \boldsymbol{z} \in Z^n.\ \mathcal{L}_a(\boldsymbol{z}, \boldsymbol{y}) - \mathcal{L}_a(\boldsymbol{z}_h, \boldsymbol{y}) \leq \gamma_a \implies d(\boldsymbol{z}, \boldsymbol{z}_h) < \epsilon_a\Big) \rho_N(\boldsymbol{y} \mid \boldsymbol{z}_h)$$

$$+ \sum_{\boldsymbol{y} \in Y^n} \mathbb{1}\Big(\forall \boldsymbol{z} \in Z^n.\ \mathcal{L}_a(\boldsymbol{z}, \boldsymbol{y}) - \mathcal{L}_a(\boldsymbol{z}_h, \boldsymbol{y}) \leq \gamma_a \implies d(\boldsymbol{z}, \boldsymbol{z}_h) < \epsilon_a\Big) \rho_N(\boldsymbol{y} \mid \boldsymbol{z}_h) - 1\Big)$$

$$\geq \max(\frac{1}{2}, (1 - \delta_a - \delta_b)) \geq 1 - \delta$$

□

5.3 Application of These Concepts to Text Manipulating Noise Sources

Many program synthesis algorithms focus on text manipulation applications [20, 35]. Building on classical research on text errors [10], I formulate a general class of noise sources and loss functions for text processing program synthesis applications and prove optimality and convergence relationships that hold between these noise sources and loss functions. These results can guide the choice of loss function given information about a noise source.

Generalized Damerau-Levenshtein Noise Source:

The generalized Damerau-Levenshtein noise source (Figure 5-1), characterizes the

$$
\begin{aligned}
\rho_{N_{gDL}}(\epsilon \mid \epsilon) &:= 1 \\
\rho_{N_{gDL}}(c \cdot s \mid \epsilon) &:= p_{\mathsf{insert}} \times \rho_{\mathsf{insert}}(c) \times \rho_{N_{gDL}}(s \mid \epsilon) \\
\rho_{N_{gDL}}(\epsilon \mid c \cdot s) &:= p_{\mathsf{delete}} \times \rho_{N_{gDL}}(\epsilon \mid s) \\
\rho_{N_{gDL}}(c_1' \cdot c_2' \cdot s' \mid c_1 \cdot c_2 \cdot s) &:= p_{\mathsf{donothing}} \times \rho_{N_{gDL}}(c_2' \cdot s' \mid c_2 \cdot s) \times \mathbb{1}\left(c_1' = c_2'\right) \\
&+ p_{\mathsf{insert}} \times \rho_{\mathsf{insert}}(c_1') \times \rho_{N_{gDL}}(c_2' \cdot s' \mid c_1 \cdot c_2 \cdot s) \\
&+ p_{\mathsf{delete}} \times \rho_{N_{gDL}}(c_1' \cdot c_2' \cdot s' \mid c_2 \cdot s) \\
&+ p_{\mathsf{substitute}} \times \rho_{\mathsf{substitute}}(c_1' \mid c_1) \times \rho_{N_{gDL}}(c_2' \cdot s' \mid c_2 \cdot s) \\
&+ p_{\mathsf{transpose}} \times \rho_{N_{gDL}}(s' \mid s) \times \mathbb{1}\left(c_1' = c_2 \text{ and } c_2' = c_1\right)
\end{aligned}
$$

Figure 5-1: Generalized Damerau-Levenshtein noise source

conditional probability $\rho_{N_{gDL}}(s' \mid s)$ of the noise source generating noisy string s' given a noise-free string s. The noise source works with four kinds of noise: 1) character insertions, 2) character deletions, 3) character substitutions, and 4) character transpositions. It is parameterized by the probabilities of applying each noise source at each text character, specifically $p_{\mathsf{insert}}, p_{\mathsf{delete}}, p_{\mathsf{substitute}}, p_{\mathsf{transpose}} \geq 0$, where $p_{\mathsf{insert}} + p_{\mathsf{delete}} + p_{\mathsf{substitute}} + p_{\mathsf{transpose}} < 1$, and by $\rho_{\mathsf{insert}}(c)$ (the probability of inserting the character c given that an insertion will happen) and $\rho_{\mathsf{substitute}}(c' \mid c)$ (the probability of substituting the original character c with c' given that a substitution will happen). I define $p_{\mathsf{donothing}} = 1 - (p_{\mathsf{insert}} + p_{\mathsf{delete}} + p_{\mathsf{substitute}} + p_{\mathsf{transpose}})$; note that $p_{\mathsf{donothing}} > 0$.

I extend the noise source $\rho_{N_{gDL}}$ to outputs $\boldsymbol{z}$ and a noisy outputs $\boldsymbol{y}$), $\boldsymbol{z} = \langle z_1, \ldots, z_n \rangle$, $\boldsymbol{y} = \langle y_1, \ldots, y_n \rangle$, by taking the product over corresponding output/noisy outputs: $\rho_{N_{gDL}}(\boldsymbol{y} \mid \boldsymbol{z}) = \prod_{1 \leq i \leq n} \rho_{N_{gDL}}(y_i \mid z_i)$.

Given s', s, and the parameters, the conditional probablity $\rho_{N_{gDL}}$ can be readily computed using a recursive computation as in Figure 5-1. Because of the parameterization, the noise source can be immediately specialized to model a range of situations, including situations in which only one of the kinds of noise is relevant.

Generalized Damerau-Levenshtein Loss Function:
Generalized Damerau-Levenshtein loss function is defined as:

$$\mathcal{L}_{gDL}(\boldsymbol{z}, \boldsymbol{y}) = -\log(\rho_{N_{gDL}}(\boldsymbol{y} \mid \boldsymbol{z}))$$

$$d_{a,b}(i,j) = \min \begin{cases} 0 & i = j = 0 \\ d_{a,b}(i-1,j) + 1 & i > 0 \ \#\text{Deletion} \\ d_{a,b}(i,j-1) + 1 & j > 0 \ \#\text{Insertion} \\ d_{a,b}(i-1,j-1) & a_i = b_j, i, j > 0 \ \#\text{Do nothing} \\ d_{a,b}(i-1,j-1) + 1 & a_i \neq b_j, i, j > 0 \ \#\text{Substitution} \\ d_{a,b}(i-2,j-2) + 1 & i, j > 1, a_i = b_{j-1}, a_{i-1} = b_j \ \#\text{Transposition} \end{cases}$$

Figure 5-2: Damerau-Levenshtein distance metric

Note that this loss function is optimal for the generalized Damerau-Levenshtein noise source.

Theorem 9. *The generalized Damerau-Levenshtein loss function $\mathcal{L}_{gDL}$ is the optimal loss function for the generalized Damerau-Levenshtein noise source N_{gDL}, given that both $\mathcal{L}_{gDL}$ and N_{gDL} are parameterized by the same parameters, insert distribution ρ_{insert}, and substitute distribution $\rho_{\mathsf{substitute}}$.*

Proof. Generalized Damerau-Levenshtein loss function is the optimal loss function for the generalized Damerau-Levenshtein noise source because it is equal to the $-\log$ of the probability measure (Section 5.1.1). □

I next present a theorem that establishes when the combination of a generalized Damerau-Levenshtein noise source and loss function is differentiating (and therefore converges in the presence of a differentiating input source) despite potential parameter mismatches. In the absence of complete or even any specific information about the parameters for the generalized Damerau-Levenshtein noise source, this theorem makes it possible to choose parameters for the generalized Damerau-Levenshtein loss function that nevertheless ensure convergence.

Definition 17. Damerau-Levenshtein Distance Metric:

$$d_{DL}(z,y) = d_{z,y}(|z|,|y|)$$

where d is defined in Figure 5-2.

Theorem 10. *Given Damerau-Levenshtein distance metric d_{DL} [10] (Definition 17) and the generalized Damerau-Levenshtein loss function $\mathcal{L}_{gDL}$ (parameterized by ρ^*_{insert}, $\rho^*_{\mathsf{substitute}}$, p^*_{insert}, p^*_{delete}, $p^*_{\mathsf{substitute}}$, $p^*_{\mathsf{transpose}}$, and $p^*_{\mathsf{donothing}}$), the generalized Damerau-Levenshtein noise source N_{gDL} (parameterized by ρ_{insert}, $\rho_{\mathsf{substitute}}$, p_{insert}, p_{delete}, $p_{\mathsf{substitute}}$, $p_{\mathsf{transpose}}$, and $p_{\mathsf{donothing}}$) is differentiating, if*

$$p_{\mathsf{insert}} > 0 \implies p^*_{\mathsf{insert}} > 0$$

$$p_{\mathsf{delete}} > 0 \implies p^*_{\mathsf{delete}} > 0$$

$$p_{\mathsf{substitute}} > 0 \implies p^*_{\mathsf{substitute}} > 0$$

$$p_{\mathsf{transpose}} > 0 \implies p^*_{\mathsf{transpose}} > 0$$

and for all characters c and c':

$$\rho_{\mathsf{insert}}(c) > 0 \implies \rho^*_{\mathsf{insert}}(c) > 0$$

$$\rho_{\mathsf{substitute}}(c \mid c') > 0 \implies \rho^*_{\mathsf{substitute}}(c \mid c') > 0$$

Proof. Consider two output vectors $\boldsymbol{z}$ and $\boldsymbol{z}_h$ of equal size. If $d_{DL}(\boldsymbol{z}, \boldsymbol{z}_h) \geq \epsilon$, then it will take at least ϵ edits (i.e., ϵ specific insertions, deletions, transpositions, and substitutions) to convert $\boldsymbol{z}$ to $\boldsymbol{z}_h$. Let E be the set of these specific edits. Now consider a new vector $\boldsymbol{y}$ of size equal to $\boldsymbol{z}_h$. Now assume $\mathcal{L}_{gDL}(\boldsymbol{z}, \boldsymbol{y}) - \mathcal{L}_{gDL}(\boldsymbol{z}_h, \boldsymbol{y}) > \gamma$. Let p_{max} and p_{min}, be the max and min of $p_{insert}, p_{transpose}, p_{delete}, p_{substitute}$, which are greater than 0.

The distance between $\boldsymbol{z}$ and $\boldsymbol{z}_h$ is greater than ϵ. Let us assume $\boldsymbol{y}$ contains m edits, then $\mathcal{L}_{gDL}(\boldsymbol{z}_h, \boldsymbol{y}) \leq m \times p_{max}$. $\mathcal{L}_{gDL}(\boldsymbol{z}_h, \boldsymbol{y}) \geq (\epsilon - m) \times p_{min}$. Therefore,

$$(\epsilon - m) \times p_{min} - m \times p_{max} > \gamma \implies \mathcal{L}_{gDL}(\boldsymbol{z}, \boldsymbol{y}) - \mathcal{L}_{gDL}(\boldsymbol{z}_h, \boldsymbol{y}) > \gamma$$

Now if we consider all vectors $\boldsymbol{z}$ of size equal to $\boldsymbol{z}_h$, such that, $d_{DL}(\boldsymbol{z}, \boldsymbol{z}_h) \geq \epsilon$, then $\forall \boldsymbol{z} \in Z^n . \mathcal{L}_{gDL}(\boldsymbol{z}, \boldsymbol{y}) - \mathcal{L}_{gDL}(\boldsymbol{z}_h, \boldsymbol{y}) > \gamma$, if and only if, $\boldsymbol{y}$ contains less than $\epsilon - m$ edits.

Note that, this can be any type of edits.

$$\rho_{N_{gDS}}(\boldsymbol{y} \text{ with at most } \epsilon - m \text{ edits} \mid \boldsymbol{z}_h) = 1 - \rho_{N_{gDS}}(\boldsymbol{y} \text{ atleast } \epsilon - m \text{ edits} \mid \boldsymbol{z}_h)$$

$$= 1 - (1 - p_{donothing})^{(\epsilon - m)}$$

Therefore, if $p_{donothing} > 0$, for any $\delta > 0$ and γ, if we pick a k, m, and an ϵ, such that,

$$\delta > (1 - p_{donothing})^{(\epsilon - m)}$$

$$(\epsilon - m) \times p_{min} - m \times p_{max} > \gamma$$

then

$$\sum_{\boldsymbol{y} \in Y^n} \mathbb{1}\Big(\forall \boldsymbol{z} \in Z^n . \mathcal{L}_{gDL}(\boldsymbol{z}, \boldsymbol{y}) - \mathcal{L}_{gDL}(\boldsymbol{z}_h, \boldsymbol{y}) \leq \gamma \implies d_{DL}(\boldsymbol{z}, \boldsymbol{z}_h) < \epsilon\Big) \rho_{N_{gDL}}(\boldsymbol{y} \mid \boldsymbol{z}_h)$$

$$= 1 - (1 - p_{donothing})^{(\epsilon - m)} \geq 1 - \delta$$

Therefore, $\rho_{N_{gDL}}$ is differentiating. □

Damerau-Levenshtein Loss Function: The Damerau-Levenshtein loss function $\mathcal{L}_{DL}(\boldsymbol{z}, \boldsymbol{x})$ uses the *Damerau-Levenshtein* distance metric (Definition 17), to measure the distance between the output from the synthesized program and the corresponding output in the noisy dataset:

$$\mathcal{L}_{DL}(\langle z_1, \ldots z_n \rangle, \langle y_1, \ldots y_n \rangle) = \sum_{i=1}^{n} L_{z_i, y_i}\Big(|z_i|, |y_i| \Big)$$

where $L_{a,b}(i, j)$ is the *Damerau-Levenshtein* distance metric [10], which counts the number of single character deletions, insertions, substitutions, or transpositions required to convert one text string into another. Because more than 80% of all human misspellings are reported to be captured by a single one of these four operations [10], the Damerau-Levenshtein loss function may be appropriate for computations that work with human-provided text input-output examples.

In addition to the previous result, I also show that the Damerau-Levenshtein loss function, in combination with a generalized Damerau-Levenshtein noise source, is differentiating (and will converge in the presence of a differentiating input source). Therefore, in the presence of any generalized Damerau-Levenshtein style noise source, using the Damerau-Levenshtein loss function will allow us to ensure convergence of the synthesis algorithm.

Theorem 11. *Given Damerau-Levenshtein distance metric d_{DL} and the Damerau-Levenshtein loss function $\mathcal{L}_{DL}$, the generalized Damerau-Levenshtein noise source N_{gDL} is differentiating.*

Proof. Consider two output vectors $\boldsymbol{z}$ and $\boldsymbol{z}_h$ of equal size. If $d_{DL}(\boldsymbol{z}, \boldsymbol{z}_h) \geq \epsilon$, then it will take at least ϵ edits (i.e., ϵ specific insertions, deletions, transpositions, and substitutions) to convert $\boldsymbol{z}$ to $\boldsymbol{z}_h$. Let E be the set of these specific edits. Now consider a new vector $\boldsymbol{y}$ of size equal to $\boldsymbol{z}_h$. Now $\mathcal{L}_{DL}(\boldsymbol{z}, \boldsymbol{y}) - \mathcal{L}_{DL}(\boldsymbol{z}_h, \boldsymbol{y}) > \gamma$, then $\boldsymbol{y}$ has at most $\epsilon - \gamma$ of these specific edits.

Now if we consider all vectors $\boldsymbol{z}$ of size equal to $\boldsymbol{z}_h$, such that, $d_{DL}(\boldsymbol{z}, \boldsymbol{z}_h) \geq \epsilon$, then $\forall \boldsymbol{z} \in Z^n . \mathcal{L}_{DL}(\boldsymbol{z}, \boldsymbol{y}) - \mathcal{L}_{DL}(\boldsymbol{z}_h, \boldsymbol{y}) > \gamma$, if and only if, $\boldsymbol{y}$ contains less than $\epsilon - \gamma$ edits. Note that this can be any type of edits.

$$\rho_{N_{gDS}}(\boldsymbol{y} \text{ with at most } \epsilon - \gamma \text{ edits} \mid \boldsymbol{z}_h) = 1 - \rho_{N_{gDS}}(\boldsymbol{y} \text{ atleast } \epsilon - \gamma \text{ edits} \mid \boldsymbol{z}_h)$$

$$= 1 - (1 - p_{donothing})^{(\epsilon - \gamma)}$$

Therefore, if $p_{donothing} > 0$, for any $\delta > 0$ and γ, if we pick a k and an ϵ, such that,

$$\delta > (1 - p_{donothing})^{(\epsilon - \gamma)}$$

then

$$\sum_{\boldsymbol{y} \in Y^n} \mathbb{1}\Big(\forall \boldsymbol{z} \in Z^n . \mathcal{L}_{DL}(\boldsymbol{z}, \boldsymbol{y}) - \mathcal{L}_{DL}(\boldsymbol{z}_h, \boldsymbol{y}) \leq \gamma \implies d_{DL}(\boldsymbol{z}, \boldsymbol{z}_h) < \epsilon\Big) \rho_{N_{gDL}}(\boldsymbol{y} \mid \boldsymbol{z}_h)$$

$$= 1 - (1 - p_{donothing})^{(\epsilon - \gamma)} \geq 1 - \delta$$

Therefore, $\rho_{N_{gDL}}$ is differentiating. $\square$

Expected Generalized Damerau-Levenshtein Loss Function: I next explore the case where a user has imperfect information about the parameters of the generalized Damerau-Levenshtein noise source. I start with a finite/infinite set $\mathcal{N}$ of generalized Damerau-Levenshein noise sources, each parameterized by potentially different values. Let $\rho_{\mathcal{N}}$ be the prior probability of a noise source within the set $\mathcal{N}$ to be the hidden noise source which corrupted the output strings.

Using the framework presented in Subsection 5.1.2, we can construct an optimal loss function (expected generalized Damerau-Levenshtein loss function) $\mathcal{L}_{egDL}$, given this imperfect information. Formally,

$$\mathcal{L}_{egDL}(\boldsymbol{z}, \boldsymbol{y}) = -\log\Big(\int_{N_{gDL} \in \mathcal{N}} \rho_{N_{gDL}}(\boldsymbol{z}, \boldsymbol{y}) \rho_{\mathcal{N}}(N_{gDL})\Big)$$

I present a theorem that establishes that the expected generalized Damerau-Levenshtein loss function $\mathcal{L}_{egDL}$ is differentiating for all noise sources in set $\mathcal{N}$. Therefore, even if we are given imperfect information about the parameters for the generalized Damerau-Levenshtein noise source, we can construct an expected generalized Damerau-Levenshtein loss function which is optimal and ensures convergence for any parameters consistent with the given imperfect information.

Theorem 12. *Given a set of Damerau-Levenshtein noise sources $\mathcal{N}$ and a prior probability distribution $\rho_{\mathcal{N}}$ over $\mathcal{N}$, Damerau-Levenshtein distance metric d_{DL} and the expected generalized Damerau-Levenshtein loss function $\mathcal{L}_{egDL}$ over $\mathcal{N}, \rho_{\mathcal{N}}$, then all generalized Damerau-Levenshtein noise sources $N_{gDL} \in \mathcal{N}$ are differentiating.*

Proof. Consider two output vectors $\boldsymbol{z}$ and $\boldsymbol{z}_h$ of equal size. If $d_{DL}(\boldsymbol{z}, \boldsymbol{z}_h) \geq \epsilon$, then it will take at least ϵ edits (i.e., ϵ specific insertions, deletions, transpositions, and substitutions) to convert $\boldsymbol{z}$ to $\boldsymbol{z}_h$. Let E be the set of these specific edits. Now consider a new vector $\boldsymbol{y}$ of size equal to $\boldsymbol{z}_h$. Now assume $\mathcal{L}_{egDL}(\boldsymbol{z}, \boldsymbol{y}) - \mathcal{L}_{egDL}(\boldsymbol{z}_h, \boldsymbol{y}) > \gamma$. Let p_{max} and p_{min}, be the max and min of $p_{insert}, p_{transpose}, p_{delete}, p_{substitute}$, for any $N'_{gDL} \in \mathcal{N}$, which are greater than 0.

The distance between $\boldsymbol{z}$ and $\boldsymbol{z}_h$ is greater than ϵ. Let us assume that $\boldsymbol{y}$ contains m edits, the $\mathcal{L}_{egDL}(\boldsymbol{z}_h, \boldsymbol{y}) \leq m \times p_{max}$. $\mathcal{L}_{egDL}(\boldsymbol{z}_h, \boldsymbol{y}) \geq (\epsilon - m) \times p_{min}$. Therefore,

$$(\epsilon - m) \times p_{min} - m \times p_{max} > \gamma \implies \mathcal{L}_{egDL}(\boldsymbol{z}, \boldsymbol{y}) - \mathcal{L}_{egDL}(\boldsymbol{z}_h, \boldsymbol{y}) > \gamma$$

Now if we consider all vectors $\boldsymbol{z}$ of size equal to $\boldsymbol{z}_h$, such that, $d_{DL}(\boldsymbol{z}, \boldsymbol{z}_h) \geq \epsilon$, then $\forall \boldsymbol{z} \in Z^n . \mathcal{L}_{egDL}(\boldsymbol{z}, \boldsymbol{y}) - \mathcal{L}_{egDL}(\boldsymbol{z}_h, \boldsymbol{y}) > \gamma$, if and only if, $\boldsymbol{y}$ contains less than $\epsilon - m$ edits. Note that this can be any type of edits.

$$\rho_{N_{gDL}}(\boldsymbol{y} \text{ with at most } \epsilon - m\text{edits} \mid \boldsymbol{z}_h) = 1 - \rho_{N_{gDL}}(\boldsymbol{y} \text{ atleast } \epsilon - m\text{edits} \mid \boldsymbol{z}_h)$$

$$= 1 - (1 - p_{donothing})^{(\epsilon - m)}$$

Therefore, if $p_{donothing} > 0$, for any $\delta > 0$ and γ, if we pick a k, m, and an ϵ, such that,

$$\delta > (1 - p_{donothing})^{(\epsilon - m)}$$

$$(\epsilon - m) \times p_{min} - m \times p_{max} > \gamma$$

then

$$\sum_{\boldsymbol{y} \in Y^n} \mathbb{1}\Big(\forall \boldsymbol{z} \in Z^n . \mathcal{L}_{egDL}(\boldsymbol{z}, \boldsymbol{y}) - \mathcal{L}_{egDL}(\boldsymbol{z}_h, \boldsymbol{y}) \leq \gamma \implies d_{DL}(\boldsymbol{z}, \boldsymbol{z}_h) < \epsilon\Big) \rho_{N_{gDL}}(\boldsymbol{y} \mid \boldsymbol{z}_h)$$

$$= 1 - (1 - p_{donothing})^{(\epsilon - m)} \geq 1 - \delta$$

Therefore, $\rho_{N_{gDL}}$ is differentiating. □

5.3.1 Connecting Theory With Experiments

I next introduce and study noise sources and loss functions introduced in Chapter 2 and Chapter 4.

n-Substitution Noise Source: The n-Substitution noise source N_{nS}, given an output vector $\langle z_1, \ldots z_n \rangle$ corrupts each string z_i independently. For each string $z = c_1 \cdots c_k$, it replaces character c_i with a random character not equal to c_i with

probability δ_{nS}.

Note that this noise source is a special case of the generalized Damerau-Levenshtein noise source.

n-Substitution Loss Function: The n-Substitution loss function $\mathcal{L}_{nS}(\boldsymbol{z}, \boldsymbol{y})$ uses per-example loss function L_{nS} that captures a weighted sum of positions where the noisy output string agrees and disagrees with the output from the synthesized program. If the synthesized program produces an output that is longer or shorter than the output in the noisy dataset, the loss function is ∞:

$$\mathcal{L}_{nS}(\langle z_1, \ldots z_n \rangle, \langle y_1, \ldots y_n \rangle) = \sum_{i=1}^{n} L_{nS}(z_i, y_i), \text{ where}$$

$$L_{nS}(z, y) = \begin{cases} \infty & |z| \neq |y| \\ \sum_{i=1}^{|z|} -\log \delta_i \text{ if } z[i] \neq y[i] & \\ \text{else } -\log(1 - \delta_i) & |z| = |y| \end{cases}$$

Note that this loss function is a linear transformation of the n-Substitution loss function proposed in Chapter 4.

The n-Substitution noise function combined with either n-Substitution loss function or Damerau-Levenshtein loss function is differentiating. Therefore, in the presence of a differentiating input source, a synthesis algorithm which uses either the n-Substitution loss function or the Damerau-Levenshtein loss function will converge if used with the n-Substitution noise source. The n-Substitution loss function is also the optimal loss function for n-Substitution noise source.

Theorem 13. *n-Substitution loss function $\mathcal{L}_{nS}$ is the optimal loss function for the n-Substitution noise source N_{nS}. Given length distance metric d_l and the n-Substitution loss function $\mathcal{L}_{nS}$, the n-Substitution noise source N_{nS} is differentiating.*

Proof.

$$-\log \rho_{N_{nS}}(\langle y_1, \ldots y_n \rangle \mid \langle z_1, \ldots z_n \rangle) = \sum_{i=1}^{n} -\log \rho_{n_{nS}}(y_i \mid z_i)$$

where:

$$-\log \rho_{n_{nS}}(s'_1 \cdot \ldots \cdot s'_k \mid s_1 \cdot \ldots \cdot s_k) = \sum_{i=1}^{n}(-\mathbb{1}(s'_i = s_i)\log(1-\delta_i)) + (-\mathbb{1}(s'_i \neq s_i)\log \delta_i)$$

Note that $\mathcal{L}_{nS} = -\log \rho_{N_{nS}}$. Hence n-Substitution loss function $\mathcal{L}_{nS}$ is the optimal loss function for n-Substitution noise source.

if $d_l(\boldsymbol{z}, \boldsymbol{z}_h) \geq 1$, then for all samples $\boldsymbol{y}$, $\mathcal{L}_{nS}(\boldsymbol{z}, \boldsymbol{y}) = \infty$. Therefore for all $\gamma > 0$, $\delta > 0$,

$$\sum_{\boldsymbol{y} \in Y^n} \mathbb{1}(\forall\, \boldsymbol{z} \in Z^n . d_l(\boldsymbol{z}, \boldsymbol{z}_h) \geq 1 \implies \mathcal{L}(\boldsymbol{z}, \boldsymbol{y}) - \mathcal{L}(\boldsymbol{z}_h, \boldsymbol{y}) = \infty > \gamma) \rho_N(\boldsymbol{y} \mid \boldsymbol{z}_h) = 1 \geq (1-\delta)$$

□

Theorem 14. *Given Damerau-Levenshtein distance metric d_{DL} and the the n-Substitution noise source N_{nS}, Damerau-Levenshtein loss function $\mathcal{L}_{DL}$ is differentiating.*

Proof. n-Substitution noise source is an instance of a generalized Damerau-Levenshtein noise source, with probability of delete, insert, and transpose set to 0. Since Damerau-Levenshtein loss function is differentiating with respect to all generalized Damerau-Levenshtein noise sources (Theorem 11), given Damerau-Levenshtein distance metric d_{DL} and the the n-Substitution noise source N_{nS}, Damerau-Levenshtein loss function $\mathcal{L}_{DL}$ is differentiating.

□

1-Delete Noise Source: The 1-Delete noise source N_{1D} given a string s corrupts it by deleting a random character with probability $\delta_{1D} < 1$. Formally:

$$\rho_{N_{1D}}(s \mid s) = 1 - \delta_{1D}$$

$$\rho_{N_{1D}}(a \cdot b \mid a \cdot c \cdot b) = \frac{1}{\mathsf{len}(a \cdot c \cdot b)}\delta_{1D}$$

where a, b are strings and c is a character.

I extend this noise source to multiple outputs and corrupted outputs by taking a

product over the corresponding output and the corrupted output:

$$\rho_{N_{1D}}(\langle y_1, \ldots y_n\rangle \mid \langle z_1, \ldots z_n\rangle) = \prod_{i=1}^{n} \rho_{n_{1D}}(y_i \mid z_i)$$

1-Delete Loss Function: The 1-Delete loss function $\mathcal{L}_{1D}(\langle z_1, \ldots z_n\rangle, \langle y_1, \ldots y_n\rangle)$ assigns loss $-\log(1-\delta_i)$ if the output z_i from the synthesized program and the dataset y_i match exactly, $-\log \delta_i$ if a single deletion enables the output from the synthesized program to match the output from the dataset, and ∞ otherwise (for $0 < \delta_i < 1$):

$$\mathcal{L}_{1D}(\langle z_1, \ldots z_n\rangle, \langle y_1, \ldots y_n\rangle) = \sum_{i=1}^{n} L_{1D}(z_i, y_i), \text{ where}$$

$$L_{1D}(z, y) = \begin{cases} -\log(1-\delta_i) & z = y \\ -\log \delta_i & a \cdot x \cdot b = z \wedge a \cdot b = y \wedge |x| = 1 \\ \infty & \text{otherwise} \end{cases}$$

Note that this loss function is a linear transformation of the 1-Delete loss function proposed in Chapter 4.

The 1-Delete noise function combined with either 1-Delete loss function or Damerau-Levenshtein loss function is differentiating. Therefore, in presence of a differentiating input source, a synthesis algorithm which uses either the 1-Delete loss function or the Damerau-Levenshtein loss function will ensure convergence, if the outputs were corrupted by 1-Delete noise source. 1-Delete loss function is also the optimal loss function for 1-Delete noise source.

Theorem 15. *1-Delete loss function $\mathcal{L}_{1D}$ is the optimal loss function for the 1-Delete noise source N_{1D}. Given DL-2 Distance Metric d_{DL2} and the 1-Delete loss function $\mathcal{L}_{1D}$, the 1-Delete noise source N_{1D} is differentiating.*

Proof.

$$-\log \rho_{N_{1D}}(\langle y_1, \ldots y_n\rangle \mid \langle z_1, \ldots z_n\rangle) = \sum_{i=1}^{n} -\log \rho_{n_{1D}}(y_i \mid z_i)$$

where $\rho_{n_{1D}}(z \mid z) = (1 - \delta_i)$ and $\rho_{n_{1D}}(y \mid z) = \delta_i$ if y has exactly one character deleted with respect to z.

Note that $\mathcal{L}_{1D} = -\log \rho_{N_{1D}}$. Hence 1-Delete Loss Function $\mathcal{L}_{1D}$ is the optimal loss function for 1-Delete Noise Source.

If $d_{DL2}(\boldsymbol{z}, \boldsymbol{z}_h) \geq 1$ then $\mathcal{L}_{1D}(\boldsymbol{z}, \boldsymbol{y}) = \infty$. Therefore for all $\gamma > 0$, $\delta > 0$,

$$\rho_N[\forall\ \boldsymbol{z} \in Z^n . d_{DL2}(\boldsymbol{z}, \boldsymbol{z}_h) \geq 1 \implies \mathcal{L}(\boldsymbol{z}, \boldsymbol{y}) - \mathcal{L}(\boldsymbol{z}_h, \boldsymbol{y}) = \infty > \gamma \mid \boldsymbol{z}_h] = 1 \geq (1 - \delta)$$

□

Theorem 16. *Given DL-2 Distance Metric d_{DL2} and the Damerau-Levenshtein loss function $\mathcal{L}_{DL}$, the 1-Delete noise source N_{1D} is differentiating.*

Proof. If $d_{DL2}(\boldsymbol{z}, \boldsymbol{z}_h) \geq m$, then $\mathcal{L}_{DL}(\boldsymbol{z}, \boldsymbol{y}) - \mathcal{L}_{DL}(\boldsymbol{z}, \boldsymbol{y}) \geq m$. Therefore for all $\gamma > 0$, $\delta > 0$,

$$\rho_N[\forall\ \boldsymbol{z} \in Z^n . d_{DL2}(\boldsymbol{z}, \boldsymbol{z}_h) \geq \gamma \implies \mathcal{L}(\boldsymbol{z}, \boldsymbol{y}) - \mathcal{L}(\boldsymbol{z}_h, \boldsymbol{y}) > \gamma \mid \boldsymbol{z}_h] = 1 \geq (1 - \delta)$$

□

The choice of loss function affects whether the noise source is differentiating or not. The noise source may reveal information identifying the hidden program with high probability, but the loss function may fail to capture this information. For example, consider that the 1-Delete noise source N_{1D} with the 1-Delete loss function $\mathcal{L}_{1D}$ (and DL-2 distance metric d_{DL2}), is differentiating. Therefore the 1-Delete noise source preserves enough information to enable successful synthesis (with the correct loss function). But consider the n-Substitution loss function $\mathcal{L}_{nS}$, which penalizes a deletion with infinite loss. With this n-Substitution loss function, the 1-Delete noise source N_{1D} is not differentiating, which eliminates any convergence guarantee:

Theorem 17. *Given n-Substitution loss function $\mathcal{L}_{nS}$, 1-Delete noise source N_{1D} is not differentiating.*

Noise source	Optimal loss function	Differentiating loss function		
		1-Delete	n-Substitution	DL
1-Delete noise source	1-Delete loss function Theorem 15	✓ Theorem 15	× Theorem 17	✓ Theorem 16
n-Substitution noise source	n-Substitution loss function Theorem 13	× Theorem 18	✓ Theorem 13	✓ Theorem 14

Figure 5-3: Summary of the optimal loss functions and differentiating loss function results.

Proof. If even a single deletion happens in $\boldsymbol{y}$, then $\mathcal{L}_{nS}(\boldsymbol{z}_h, \boldsymbol{y}) = \infty$. The probability of no deletions is equal to $(1 - \delta_i)^n$ which decreases with n, therefore in this case 1-Delete Noise Source is non-differentiating. □

Theorem 18. *Given 1-Delete loss function $\mathcal{L}_{1D}$, n-Substitution noise source N_{nS} is not differentiating.*

Proof. If even a single substitution happens in $\boldsymbol{y}$, then $\mathcal{L}_{1D}(\boldsymbol{z}_h, \boldsymbol{y}) = \infty$. The probability of no substitutions is equal to $(1 - \delta_{nS})^n$ which decreases with n, therefore in this case n-Substitution Noise Source is non-differentiating. □

In Chapter 4, we empirically show that, for successful noisy synthesis, the n-Substitution noise source requires all input-output examples to be correct (i.e., cannot tolerate any noise) when the noise is introduced by the 1-Delete noise source. Theorem 17 is consistent with these experimental results.

Figure 5-3 summarizes my theoretical results. The first column presents the name of the noise source. The next column presents the optimal loss function for the given noise source. The next three columns presents if the loss functions 1-Delete, n-Substitution, and Damerau-Levenstein is differentiating for the given noise source.

Connections to the Empirical Results: In Chapter 4, I empirically evaluated the Damerau-Levenshtein and 1-Delete loss functions with the 1-Delete noise source.

Both the 1-Delete loss function and Damerau-Levenshtein loss function are able to synthesize the correct program in presence of some, and in some cases a remarkable amount of, noise. In comparison with the (suboptimal for 1-Delete noise source) Damerau-Levenshtein loss function, our noisy synthesis algorithm tolerates datasets with more noise when using the (optimal for 1-Delete noise source) 1-Delete loss function. Even when all input-output examples were corrupted, the algorithm was able to use the 1-Delete loss function to synthesize the correct program. Because 1-Delete loss function is the optimal loss function in presence of 1-Delete noise source, it has a higher probability of synthesizing the correct program than the suboptimal in this context Damerau-Levenshtein loss function.

In Chapter 4, I also empirically evaluate the n-Substitution loss function in presence of the n-Substitution noise source. Consistent with my results, this technique was able to synthesize the correct answer over datasets corrupted by n-Substitution noise source.

5.4 Experimental Results

I used my noisy program syntheis algorithm (Chapter 3) to measure the convergence for a large class of loss functions and noise sources using noisy versions of program synthesis problems from the SyGus 2018 benchmark set [1]. Each problem in this benchmark set is defined by a (noise free) data set of input/output examples defining the synthesis problem. Each problem comes with a known ground-truth solution to the synthesis problem. There are multiple versions of each problem, with the versions differing in the size of the data set. All of my experiments use the long data set provided with each problem (but, as described below, use random sampling from this long data set to obtain smaller noisy synthesis problems). I only present results for benchmarks on which the synthesis algorithm terminates within a timeout of 120 seconds for each synthesis task (Chapter 4).

Given a synthesis problem, including a data set of noise-free input/output examples, and parameters that specify a generalized Damerau-Levenshtein noise source, I systematically generate noisy synthesis problems of varying sizes as follows. Using a

uniform probability distribution, I sample (with replacement) the specified number of input/output examples and apply the specified generalized Damerau-Levenshtein noise source to the sampled input/output examples to obtain a noisy data set. I then apply my noisy program synthesis algorithm to solve the resulting noisy program synthesis problem and compare the resulting synthesized program with the known ground truth solution to the corresponding noise-free benchmark synthesis problem to check if they are equivalent.

My experiments start with noisy program synthesis problems of size one (with a single input/output example), then increase the size by one until termination as defined below. For each problem size we obtain 100 noisy program synthesis problems via the sampling and noise source application process described above. I then solve each sampled noisy program synthesis problem. I continue generating and solving noisy program synthesis problems of increasing size, terminating when 95 of the 100 problems synthesize a program equivalent to the original ground truth program. I report this problem size.

Damerau-Levenshtein distance metric vs optimal loss function: Figure 5-4 presents results from one of these experiments for benchmarks phone-long, phone-1-long, phone-2-long, phone-5-long, phone-6-long, phone-7-long, and phone-8-long. For each problem I consider two loss functions, specifically the optimal loss function for the noise source used to generate the data set and the traditional Damerau-Levenshtein distance metric (which counts the minimum number of edits required to transform the output from the synthesized program into the output from the noisy data set). The first column of the figure presents the four parameters of the generalized Damerau-Levenshtein noise source in the form $p_{\mathsf{i}}, p_{\mathsf{d}}, p_{\mathsf{s}}$, and p_{t}. For each text character, the noise source inserts a new character with probability p_{i} (p_{insert}), deletes the character with probability p_{d} (p_{delete}), substitutes a new character for the current character with probability p_{s} ($p_{\mathsf{substitute}}$), or swaps the position of the current character and the next character with probability p_{t} ($p_{\mathsf{transpose}}$) (Section 5.3). The noise source uniformly samples a character from set $[a - z0 - 9]$ for insertions and substitutions. Note that for some of the experimental parameters, the noise source corrupts each character

with probability 0.4. For each benchmark, the figure has two columns. The first column (DL) presents the minimum problem size for which the Damerau-Levenshtein distance metric synthesizes a program equivalent to the original ground truth program for 95 of the 100 generated noisy program synthesis problems of that size. The second column (O) presents the corresponding minimum problem size for the optimal loss function for the noise source used to generate the noisy program synthesis problems.

I present results for two classes of problems - uniform problems, in which the edit probabilities are the same for all four kinds of edits and point problems, in which all edit probabilities are zero except for a single kind of edit. Most problems require only (typically small) single digit problem sizes to meet the 95 out of 100 threshold for both the Damerau-Levenshtein distance metric and the optimal loss function. Exceptions are uniform problems with large edit probabilities (both Damerau-Levenshtein distance metric and optimal loss function) and point problems with the delete edit, in which the 95 out of 100 threshold is significantly larger for the Damerau-Levenshtein distance metric than for the optimal loss function – the Damerau-Levenshtein distance metric assigns equal weight to deletions, insertions, substitutions, and transpositions. Equally weighting all of these edits enables multiple different explanations for corrupted outputs, which in turn can generate ambiguity in the correct program for small data set sizes. I also note that for some problems the Damerau-Levenshtein distance metric reaches 95 out of 100 for smaller data set sizes than the optimal loss function, which we attribute to fluctuations caused by the randomized sampling that generated the noisy program synthesis problem.

Together, these results highlight 1) the effectiveness of noisy program synthesis as formulated in this paper, 2) the effectiveness of the Damerau-Levenshtein distance metric for a large range of noisy text synthesis problems, and 3) the effectiveness of the optimal loss function for specific targeted problems. These results also indicate that, in general, in absence of information about the noise source, Damerau-Levenshtein distance metric is an effective loss function for program synthesis over noisy strings.

Figure 5-5, 5-6, and 5-7 presents the corresponding results for bikes-long, initials-long, and first-name long; the remaining benchmarks exceed the 120 seconds timeout

Noise source parameters				Minimum problem size													
				phone		phone-1		phone-2		phone-5		phone-6		phone-7		phone-8	
p_{i}	p_{d}	p_{s}	p_{t}	DL	O	DL	O	DL	O	DL	O	DL	O	DL	O	DL	O
0.025	0.025	0.025	0.025	3	3	3	4	3	3	5	5	7	6	7	7	2	2
0.05	0.05	0.05	0.05	6	6	5	4	6	6	6	7	10	8	7	7	2	2
0.075	0.075	0.075	0.075	7	8	7	8	7	7	10	7	11	11	9	7	2	3
0.1	0.1	0.1	0.1	12	15	10	12	10	13	15	13	15	15	11	7	4	4
0.05	0.0	0.0	0.0	2	2	2	1	2	1	4	4	6	6	5	6	1	1
0.1	0.0	0.0	0.0	2	2	2	2	2	1	4	4	6	6	5	5	2	2
0.15	0.0	0.0	0.0	2	2	2	2	3	2	5	4	6	5	5	6	2	1
0.2	0.0	0.0	0.0	4	2	3	2	3	2	4	5	6	5	6	5	3	2
0.0	0.05	0.0	0.0	5	3	3	2	3	3	6	5	8	5	7	5	2	1
0.0	0.1	0.0	0.0	6	3	6	3	7	3	7	5	11	7	7	5	2	1
0.0	0.15	0.0	0.0	9	4	13	3	12	3	13	5	17	6	10	7	3	2
0.0	0.2	0.0	0.0	19	4	22	4	19	4	21	6	25	6	21	7	6	2
0.0	0.0	0.05	0.0	2	1	2	1	2	1	5	4	6	5	6	5	1	1
0.0	0.0	0.1	0.0	2	2	2	1	3	2	5	4	6	6	6	5	1	2
0.0	0.0	0.15	0.0	3	1	3	2	3	2	6	4	6	6	5	6	2	1
0.0	0.0	0.2	0.0	2	4	3	4	3	3	6	4	6	6	7	6	2	2
0.0	0.0	0.0	0.05	2	2	2	1	2	2	5	5	5	5	5	5	1	1
0.0	0.0	0.0	0.1	2	4	2	2	3	2	6	5	6	5	4	5	1	1
0.0	0.0	0.0	0.15	4	4	5	4	2	4	6	4	6	5	6	5	2	1
0.0	0.0	0.0	0.2	7	5	7	6	6	5	5	4	6	6	5	7	2	1

Figure 5-4: Minimum problem size for Damerau-Levenshtein distance metric vs optimal loss function.

limit. For these three experiments, I only experiment with even problem sizes. I report the minimum even problem size for which the Damerau-Levenshtein distance metric synthesizes a program equivalent to the original ground truth program for 90 of the 100 generated noisy program synthesis problems of that size. I also report this metric for the optimal loss function.

Uniformly parameterized loss functions and noise sources: I next present the minimum problem size for experiments in which both loss function and noise sources are uniformly parameterized. Figure 5-8 presents results for these experiments for phone-long, phone-1-long, phone-2-long, phone-5-long, phone-6-long, phone-7-long, and phone-8-long synthesis problems. For each problem, I present results for four loss functions and four noise sources. The first column of the figure presents the value for the parameters of the generalized Damerau-Levenshtein noise source in the form $p_{\mathsf{i}}, p_{\mathsf{d}}, p_{\mathsf{s}}$, and p_{t}. In this case, $p_{\mathsf{i}} = p_{\mathsf{d}} = p_{\mathsf{s}} = p_{\mathsf{t}}$ is equal to the parameter value. For each loss function parameter value, the figure has four columns. These columns present the minimum problem size for which the generalized Damerau-Levenshtein loss function, uniformly parameterized by values $0.025, 0.05, 0.075$, and 0.1 (i.e., p_{insert}, p_{delete}, $p_{\mathsf{substitute}}$, and $p_{\mathsf{transpose}}$ are equal and set to either $0.025, 0.05, 0.075$, or 0.1), synthesizes a program equivalent to the original ground truth program for 95 of the 100 generated noisy program synthesis problems of that size. The minimum problem

Noise source parameters				Minimum problem size	
p_{i}	p_{d}	p_{s}	p_{t}	Damerau-Levenshtein	optimal loss function
0.025	0.025	0.025	0.025	6	6
0.05	0.05	0.05	0.05	8	8
0.075	0.075	0.075	0.075	12	10
0.1	0.1	0.1	0.1	16	22
0.05	0.0	0.0	0.0	4	6
0.1	0.0	0.0	0.0	4	4
0.15	0.0	0.0	0.0	4	6
0.2	0.0	0.0	0.0	6	4
0.0	0.05	0.0	0.0	4	6
0.0	0.1	0.0	0.0	10	6
0.0	0.15	0.0	0.0	12	6
0.0	0.2	0.0	0.0	22	6
0.0	0.0	0.05	0.0	4	4
0.0	0.0	0.1	0.0	4	4
0.0	0.0	0.15	0.0	6	6
0.0	0.0	0.2	0.0	6	4
0.0	0.0	0.0	0.05	4	6
0.0	0.0	0.0	0.1	4	6
0.0	0.0	0.0	0.15	6	6
0.0	0.0	0.0	0.2	6	6

Figure 5-5: Minimum problem size for Damerau-Levenshtein and optimal loss function for bikes-long SyGuS benchmark.

size, in this case, is largely dependent on the level noise in the dataset (parameter of the noise source). Overall, the results indicate that the uniformly parameterized generalized Damerau-Levenshtein loss function is largely robust against differences between its parameters and the noise source's parameters.

Point parameterized loss functions and noise sources: Figure 5-9 presents the minimum problem size for experiments in which both the loss function and the noise source are point parameterized, i.e., all except one of their parameters is 0. The figure presents results for phone-long, phone-1-long, phone-2-long, phone-5-long, phone-6-long, phone-7-long, and phone-8-long synthesis problems. The *Noise source parameter* column presents the parameter value for the noise source. In these experiments, based on which parameter is non-zero, the noise source will corrupt using only either insertions, deletions, substitutions, or transpositions. I present results

Noise source parameters				Dataset size	
p_{i}	p_{d}	p_{s}	p_{t}	Damerau-Levenshtein	optimal loss function
0.025	0.025	0.025	0.025	4	4
0.05	0.05	0.05	0.05	6	6
0.075	0.075	0.075	0.075	10	8
0.1	0.1	0.1	0.1	14	20
0.05	0.0	0.0	0.0	4	4
0.1	0.0	0.0	0.0	4	4
0.15	0.0	0.0	0.0	4	4
0.2	0.0	0.0	0.0	4	4
0.0	0.05	0.0	0.0	4	4
0.0	0.1	0.0	0.0	8	4
0.0	0.15	0.0	0.0	12	4
0.0	0.2	0.0	0.0	36	4
0.0	0.0	0.05	0.0	4	4
0.0	0.0	0.1	0.0	4	4
0.0	0.0	0.15	0.0	4	4
0.0	0.0	0.2	0.0	6	4
0.0	0.0	0.0	0.05	4	4
0.0	0.0	0.0	0.1	4	6
0.0	0.0	0.0	0.15	8	6
0.0	0.0	0.0	0.2	8	6

Figure 5-6: Minimum problem size for Damerau-Levenshtein and optimal loss function for initials-long SyGuS benchmark.

Noise source parameters				Minimum problem size	
p_{i}	p_{d}	p_{s}	p_{t}	Damerau-Levenshtein	optimal loss function
0.025	0.025	0.025	0.025	2	4
0.05	0.05	0.05	0.05	4	4
0.075	0.075	0.075	0.075	6	8
0.1	0.1	0.1	0.1	8	14
0.05	0.0	0.0	0.0	2	2
0.1	0.0	0.0	0.0	2	2
0.15	0.0	0.0	0.0	2	2
0.2	0.0	0.0	0.0	4	2
0.0	0.05	0.0	0.0	4	2
0.0	0.1	0.0	0.0	4	2
0.0	0.15	0.0	0.0	10	2
0.0	0.2	0.0	0.0	14	4
0.0	0.0	0.05	0.0	2	2
0.0	0.0	0.1	0.0	2	2
0.0	0.0	0.15	0.0	2	2
0.0	0.0	0.2	0.0	2	2
0.0	0.0	0.0	0.05	2	2
0.0	0.0	0.0	0.1	2	4
0.0	0.0	0.0	0.15	2	4
0.0	0.0	0.0	0.2	2	4

Figure 5-7: Minimum problem size for Damerau-Levenshtein and optimal loss function for firstname-long SyGuS benchmark.

Noise source parameter	Problem size: Loss function parameter 0.025	0.05	0.075	0.1
0.025	4	4	3	4
0.05	6	5	6	5
0.075	7	8	8	8
0.1	11	14	13	15

(a) phone-long

Noise source arameter	Problem size: Loss function parameter 0.025	0.05	0.075	0.1
0.025	3	4	4	3
0.05	6	5	6	6
0.075	7	7	9	10
0.1	10	11	12	14

(b) phone-1-long

Noise source parameter	Problem size: Loss function parameter 0.025	0.05	0.075	0.1
0.025	3	4	3	4
0.05	5	4	6	6
0.075	7	8	7	8
0.1	10	11	15	14

(c) phone-2-long

Noise source parameter	Problem size: Loss function parameter 0.025	0.05	0.075	0.1
0.025	5	5	7	4
0.05	6	7	6	7
0.075	6	6	8	7
0.1	11	11	11	12

(d) phone-5-long

Noise source parameter	Problem size: Loss function parameter 0.025	0.05	0.075	0.1
0.025	7	7	5	6
0.05	10	9	7	8
0.075	10	10	11	10
0.1	15	15	13	17

(e) phone-6-long

Noise source parameter	Problem size: Loss function parameter 0.025	0.05	0.075	0.1
0.025	6	5	6	5
0.05	7	7	8	6
0.075	8	8	9	6
0.1	15	10	11	12

(f) phone-7-long

Noise source parameter	Problem size: Loss function parameter 0.025	0.05	0.075	0.1
0.025	2	2	2	2
0.05	2	2	3	3
0.075	2	3	3	3
0.01	4	3	4	4

(g) phone-8-long

Figure 5-8: Performance of uniformly parameterized generalized Damerau-Levenshtein loss functions in presence of uniformly parameterized Damerau-Levenshtein noise sources.

for each type of corruption. Columns under the *Insertion* header present results for experiments in which both the noise source and the loss function have their insertion parameter as the non-zero parameter. Similarly, columns under the *Deletion*, *Substi-*

tution, and *Transposition* headers present results for experiments in which both the noise source and the loss function have their deletion, substitution, and the transposition parameters, respectively, as the non-zero parameters. Each header contains four columns. These columns present the minimum problem size for loss functions with their non-zero parameter set to 0.1, 0.2, 0.3, and 0.4, respectively.

Bench--mark	Noise source param	Minimum problem size															
		Insertion				Deletion				Substitution				Transposition			
		Loss function				Loss function				Loss function				Loss function			
		0.1	0.2	0.3	0.4	0.1	0.2	0.3	0.4	0.1	0.2	0.3	0.4	0.1	0.2	0.3	0.4
phone	0.1	2	2	2	2	4	3	3	3	2	1	1	2	3	2	3	2
	0.2	2	2	2	2	5	4	4	4	3	3	3	3	3	6	4	6
	0.3	2	2	2	4	4	5	6	6	5	5	6	4	11	6	6	7
	0.4	3	3	3	6	7	6	7	7	6	7	9	8	9	10	13	11
phone-1	0.1	2	1	2	2	3	4	4	3	2	1	2	1	3	2	2	2
	0.2	2	2	2	3	4	5	4	5	3	3	2	3	5	6	6	4
	0.3	2	2	2	2	6	5	5	6	5	5	6	7	8	7	6	6
	0.4	2	2	3	5	8	6	7	8	6	7	7	8	11	11	10	11
phone-2	0.1	2	1	2	2	3	3	3	3	2	2	2	2	3	3	2	2
	0.2	2	2	2	2	4	4	5	4	3	4	5	3	6	6	7	5
	0.3	2	2	2	4	5	5	6	7	5	5	6	7	7	6	6	8
	0.4	2	2	2	5	7	7	8	6	7	7	8	7	11	9	11	9
phone-5	0.1	5	4	4	4	6	5	6	5	4	5	5	4	5	5	5	4
	0.2	4	4	5	5	6	6	6	7	4	4	5	4	4	5	5	5
	0.3	5	4	5	4	8	8	7	8	4	4	4	5	4	5	5	5
	0.4	5	4	4	5	8	9	10	8	5	5	4	5	4	4	5	3
phone-6	0.1	5	5	5	6	7	6	6	7	5	5	5	6	5	5	5	6
	0.2	5	4	5	5	7	7	7	7	6	5	6	4	5	5	5	6
	0.3	5	5	4	6	9	8	8	10	5	5	6	6	5	5	5	4
	0.4	5	6	5	6	12	11	10	11	6	7	7	7	5	6	6	6
phone-7	0.1	5	5	5	5	6	5	6	6	5	5	5	6	5	5	5	6
	0.2	5	4	5	5	7	7	7	6	6	5	6	5	5	5	5	6
	0.3	5	5	4	7	9	6	8	8	5	5	5	6	5	5	5	4
	0.4	5	6	5	6	8	8	10	7	6	6	6	6	5	6	6	6
phone-8	0.1	1	1	1	1	2	2	2	2	1	1	1	2	1	1	1	1
	0.2	2	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1
	0.3	2	2	2	2	2	2	2	2	2	2	2	3	1	1	1	1
	0.4	2	2	2	2	3	2	2	3	3	2	3	4	2	1	2	2

Figure 5-9: Performance of point parameterized generalized Damerau-Levenshtein loss functions in presence of point parameterized generalized Damerau-Levenshtein noise sources.

For point parameterized noise source corrupting strings using deletions, compared to the Damerau-Levenstein distance metric (Figure 5-4), the point parameterized generalized Damerau-Levenstein loss functions require significantly smaller problem sizes to meet the 95 out of 100 threshold. Even non-optimal point parameterized generalized loss function require significantly smaller problem sizes to meet the 95 out of 100 threshold.

Uniformly parameterized loss functions and point parameterized noise sources: Figures 5-10, 5-11, 5-12, 5-13, 5-14, 5-15, and 5-16 presents experiments with point parameterized noise source with uniformly parameterized loss functions for the phone-long, phone-1-long, phone-2-long, phone-5-long, phone-6-long, phone-

Type of noise	Noise source parameter	Dataset size			
		Loss function parameter			
		0.025	0.05	0.075	0.1
Insertions	0.05	3	3	2	2
	0.1	4	3	3	3
	0.15	4	4	4	4
	0.2	6	6	7	6
Deletions	0.05	4	3	4	3
	0.1	8	7	7	4
	0.15	9	10	7	10
	0.2	25	28	28	23
Substitutions	0.05	2	3	3	2
	0.1	4	3	3	5
	0.15	4	4	4	4
	0.2	5	4	5	6
Transposition	0.05	2	2	3	2
	0.1	3	3	5	4
	0.15	4	5	4	5
	0.2	6	6	6	9

Figure 5-10: Performance of uniformly parameterized generalized Damerau-Levenshtein loss functions in presence of point parameterized generalized Damerau-Levenshtein noise sources for phone-long synthesis problem.

Type of noise	Noise source parameter	Minimum problem size			
		Loss function parameter			
		0.025	0.05	0.075	0.1
Insertions	0.05	3	3	3	3
	0.1	4	4	3	4
	0.15	5	4	5	4
	0.2	6	7	7	7
Deletions	0.05	4	4	4	4
	0.1	7	7	5	5
	0.15	12	13	11	10
	0.2	24	20	28	23
Substitutions	0.05	3	3	3	2
	0.1	3	3	4	3
	0.15	5	4	4	4
	0.2	5	5	4	5
Transposition	0.05	3	2	3	3
	0.1	4	4	4	3
	0.15	5	5	5	6
	0.2	6	8	8	9

Figure 5-11: Performance of uniformly parameterized generalized Damerau-Levenshtein loss functions in presence of point parameterized generalized Damerau-Levenshtein noise sources for phone-1-long synthesis problem.

Type of noise	Noise source parameter	Minimum problem size			
		Loss function parameter			
		0.025	0.05	0.075	0.1
Insertions	0.05	3	2	3	2
	0.1	3	4	3	3
	0.15	5	5	5	5
	0.2	6	5	6	6
Deletions	0.05	5	3	3	4
	0.1	6	7	7	6
	0.15	11	12	11	8
	0.2	23	25	29	30
Substitutions	0.05	3	3	3	2
	0.1	3	4	3	3
	0.15	5	5	4	4
	0.2	4	5	5	5
Transposition	0.05	3	3	2	3
	0.1	3	5	3	2
	0.15	4	5	7	5
	0.2	8	6	8	7

Figure 5-12: Performance of uniformly parameterized generalized Damerau-Levenshtein loss functions in presence of point parameterized generalized Damerau-Levenshtein noise sources for phone-2-long synthesis problem.

7-long, and phone-8-long synthesis problems. The first column presents the type of corruption the noise source will introduce. The next column presents the value of the non-zero parameter for the noise source. The next four columns present the minimum problem size for uniformly parameterized loss functions with parameters set to $0.025, 0.5, 0.075$, and 0.1, respectively. Overall, the performance of the uniformly parameterized loss function is similar to the Damerau-Levenshtein distance metric (Figure 5-4).

I also ran experiments with uniformly parameterized noise sources and point parameterized loss functions. This combination is not differentiating and therefore effectively diables the ability of the synthesis algorithm to identify correct programs — intuitively, point loss functions only extract information from strings corrupted with the kind of edit they are designed to measure.

These experimental results work with text problems with inputs constructed by randomly sampling input-output pairs from a set of benchmark input-output exam-

Type of noise	Noise source parameter	Minimum problem size			
		Loss function parameter			
		0.025	0.05	0.075	0.1
Insertions	0.05	4	4	5	4
	0.1	5	5	5	5
	0.15	4	5	5	5
	0.2	6	6	6	5
Deletions	0.05	7	5	5	5
	0.1	8	6	9	7
	0.15	12	12	10	10
	0.2	12	15	20	18
Substitutions	0.05	6	5	4	4
	0.1	4	4	5	4
	0.15	5	5	5	4
	0.2	5	6	5	5
Transposition	0.05	5	3	4	4
	0.1	5	4	4	4
	0.15	4	5	4	4
	0.2	6	4	5	5

Figure 5-13: Performance of uniformly parameterized generalized Damerau-Levenshtein loss functions in presence of point parameterized generalized Damerau-Levenshtein noise sources for phone-5-long synthesis problem.

ples. For completeness, I next show that, if we have two nonequivalent programs and an input-output example in the benchmark examples on which the nonequivalent programs produce different outputs, then the input source constructed by randomly sampling examples from the benchmark examples is differentiating.

Theorem 19. *Let G be a finite set of programs and let $\mathcal{I}$ be a finite set of inputs. Let d be a distance metric. Let ρ_I be a probability distribution over $\mathcal{I}$, such that, for all $x \in \mathcal{I}$, $\rho_I(x) > 0$, i.e., each input in $\mathcal{I}$ has a positive probability of being sampled. Let ρ_i be an input source, such that,*

$$\rho_i(\langle x_1, \ldots x_n \rangle \mid n) = \prod_{j=1}^{n} \rho_I(x_j)$$

i.e., when sampling an input vector $\boldsymbol{x} = \langle x_1, \ldots x_n \rangle$ from the input distribution ρ_i, all inputs $x_1, \ldots x_n$ are i.i.d. and are sampled from a distribution ρ_I. If for any two non-equivalent programs p and p' in G ($p, p' \in G, p \not\approx p'$) there exists an input $x \in \mathcal{I}$,

Type of noise	Noise source parameter	Minimum problem size			
		Loss function parameter			
		0.025	0.05	0.075	0.1
Insertions	0.05	6	5	5	4
	0.1	6	6	6	6
	0.15	7	6	6	5
	0.2	6	5	6	6
Deletions	0.05	7	7	8	7
	0.1	10	8	9	9
	0.15	13	14	15	16
	0.2	29	26	23	30
Substitutions	0.05	5	5	6	5
	0.1	5	6	6	6
	0.15	5	6	6	6
	0.2	6	6	6	7
Transposition	0.05	5	5	5	6
	0.1	5	5	5	5
	0.15	5	6	6	5
	0.2	6	6	6	6

Figure 5-14: Performance of uniformly parameterized generalized Damerau-Levenshtein loss functions in presence of point parameterized generalized Damerau-Levenshtein noise sources for phone-6-long synthesis problem.

such that, $d(p(x), p'(x)) > 0$*, then the input source* ρ_I^S *is differentiating.*

Proof. Let p_h be a program in G, $\delta > 0$ be a probability tolerance, and $\epsilon > 0$ be a distance value. Let $G' = G - G_{p_h}$ be the set of programs not equivalent to p_h. Let $|G'| = m$.

Let $\boldsymbol{x}_m = \langle x_1, \ldots x_m \rangle$ be a set of inputs, such that, for all programs $p \in G'$, there exists at least one input x_j, such that, $d(p_h[x_j], p[x_j]) > 0$. Let $\epsilon_i = \min_{p \in G'} d(p_h[\boldsymbol{x}_m], p[\boldsymbol{x}_m])$ and $\delta_i = \min_{j=1\ldots m} \rho_I(x_j)$. Let m_0 and n_0 be natural numbers such that $m_0 \geq \frac{\epsilon}{\epsilon_i}$, for $n \geq m n_0$,

$$\sum_{j=0}^{m_0} C_j^{n_0} \delta_i^j (1 - \delta_i)^{n_0 - j} \leq \frac{\delta}{m}$$

Let $\boldsymbol{x}$ be an input of size n, which contains at least m_i, n_i,

$$\sum_{\boldsymbol{x} \in X^n} \mathbb{1}(\forall p \in G'.\ d(p_h[\boldsymbol{x}], p[\boldsymbol{x}]) > \epsilon) \rho_i(\boldsymbol{x} \mid n)$$

Type of noise	Noise source parameter	Minimum problem size			
		Loss function parameter			
		0.025	0.05	0.075	0.1
Insertions	0.05	6	6	5	5
	0.1	5	6	5	6
	0.15	5	6	5	7
	0.2	6	7	7	6
Deletions	0.05	6	7	7	6
	0.1	6	8	8	8
	0.15	11	10	12	13
	0.2	15	17	20	23
Substitutions	0.05	5	5	6	5
	0.1	6	6	5	5
	0.15	6	6	6	5
	0.2	6	6	6	5
Transposition	0.05	5	6	5	5
	0.1	5	5	4	5
	0.15	6	5	6	5
	0.2	5	6	5	5

Figure 5-15: Performance of uniformly parameterized generalized Damerau-Levenshtein loss functions in presence of point parameterized generalized Damerau-Levenshtein noise sources for phone-7-long synthesis problem.

$$\geq \sum_{\boldsymbol{x} \in X^n} \mathbb{1}(\forall p \in G'.\ d(p_h[\boldsymbol{x}], p[\boldsymbol{x}]) > m_i \epsilon_i) \rho_i(\boldsymbol{x} \mid n)$$

$$\geq \sum_{\boldsymbol{x} \in X^n} \mathbb{1}(\text{each input } x_1, \ldots x_m \text{ occur at least } m_0 \text{ times in } \boldsymbol{x}) \rho_i(\boldsymbol{x} \mid n)$$

$$\geq \sum_{\boldsymbol{x} \in X^n} \mathbb{1}(\text{each input } x_1, \ldots x_m \text{ occur at least } m_0 \text{ times in } \boldsymbol{x}) \rho_i(\boldsymbol{x} \mid n)$$

$$\geq \sum_{\boldsymbol{x} \in X^n} \mathbb{1}(\text{input } x_j, \ldots x_m \text{ occur at least } m_0 \text{ times in } \boldsymbol{x})$$

$$\geq (1 - \sum_{j=0}^{m_0} C_j^{n_0} \delta_i^j (1 - \delta_i)^{n_0 - j})^m \geq (1 - \frac{\delta}{m})^m \geq 1 - \delta$$

Therefore, ρ_i is differentiating. □

These experimental results work with finite sets of programs (generated by applying a finite bound to the SyGuS DSL). The following theorem applies in this case.

Theorem 20. *Let G be a finite set of string processing programs. Given a finite list of inputs $\mathcal{I}$ and Damerau-Levenshtein distance metric d_{DL}, an input source, which*

Type of noise	Noise source parameter	Minimum problem size			
		Loss function parameter			
		0.025	0.05	0.075	0.1
Insertions	0.05	2	2	2	2
	0.1	3	3	2	3
	0.15	3	3	3	3
	0.2	4	4	3	5
Deletions	0.05	2	2	2	2
	0.1	2	3	2	3
	0.15	3	3	4	3
	0.2	3	3	4	5
Substitutions	0.05	1	1	1	1
	0.1	1	2	2	1
	0.15	2	2	2	2
	0.2	2	2	2	2
Transposition	0.05	1	1	1	1
	0.1	1	1	1	1
	0.15	1	2	1	1
	0.2	1	1	1	1

Figure 5-16: Performance of uniformly parameterized generalized Damerau-Levenshtein loss functions in presence of point parameterized generalized Damerau-Levenshtein noise sources for phone-8-long synthesis problem.

randomly samples inputs from $\mathcal{I}$ with uniform probability, is differentiating if for any two non-equivalent programs $p, p' \in G$, there exists at least one input $x \in \mathcal{I}$, such that, output strings $p(x)$ and $p'(x)$ disagree on at least one character.

Proof. If $p(x)$ and $p'(x)$ disagree on at least one character then $d_{DL}(p(x), p'(x)) > 0$. Rest follows from Theorem 19. □

5.5 Discussion

My formulation of the noisy program synthesis problem enables us to precisely and formally define the concepts of an optimal loss function and convergence. Building on these concepts, I characterize a large range of potential noise sources, corresponding optimal loss functions, and the conditions under which we can expect noisy program synthesis algorithms to converge. These results provide insight into the empirical results presented in Chapter 4 and can help guide future efforts in this area.

Chapter 6

Synthesis Using Abstraction Refinement Based Optimization

I introduce a new synthesis algorithm based on abstraction refinement to solve noisy program synthesis. This algorithm builds upon the concept of an abstract finite tree automaton (AFTA).

An abstract finite tree automaton substitutes abstract values in place of concrete values in an CFTA (Section 3). The AFTA promotes the construction of a coarser partition of the program space (compared to CFTA) by grouping programs with different concrete output values but the same abstract output values together into the same partition. This allows me to compute a lower bound of the objective function over all programs grouped together into the same abstract partition. By iterating over all partitions and comparing the complexity of their simplest programs combined with the lower bound of each partition's objective function values, the synthesis algorithm selects a candidate partition and a possible optimal program. I prove that the lower bound objective function value of this candidate program is not greater than the objective function value of any program in the program space. If the objective function value of the candidate program is equal to the lower bound, the algorithm halts and returns this program. I prove that this program minimizes the objective function.

If the objective function value of the candidate program is greater than the lower bound, then there may exist another program which better fits the noisy dataset.

Since the candidate program's objective function value is greater than the lower bound, there exists at least one input-output example such that the candidate program's loss/regularizer weight on this counterexample is greater than the lower bound loss/regularizer weight (over all programs in the candidate partition) on this counterexample. I use this counterexample to refine the abstractions and further partition the program space into smaller partitions. After the refinement process, I rebuild the finite tree automaton. My algorithm refines the abstractions in a manner, such that, the candidate program from the previous iteration is in a different partition in this new automaton. The lower bound loss value of this new partition is greater than the lower bound loss of candidate partition from the previous iteration. I prove that we can repeat this process and it will eventually synthesize a candidate program which minimizes the objective function.

6.1 Abstractions

I construct an abstract version of the CFTA by associating abstract values with each symbol. I assume that the abstract values are represented as conjunctions of predicates of the form $f(s)\ \mathsf{op}\ c$, where s is a symbol in the given DSL, f is a function, op is an operator, and c is a constant.

Universe of predicates: My algorithm is parameterized by a universe $\mathcal{U}$ of predicates. These are used by my algorithm to construct abstractions for values generated by sub-expressions in the DSL. The universe $\mathcal{U}$ is specified using a family of functions $\mathcal{F}$, a set of operators $\mathcal{O}$, and a set of constants $\mathcal{C}$. All predicates in the universe $\mathcal{U}$, except predicates true and false, are of the form $f(s)\ \mathsf{op}\ c$, where $f \in \mathcal{F}$, $\mathsf{op} \in \mathcal{O}$, $c \in \mathcal{C}$, and s is a symbol in the DSL. I assume $\mathcal{F}$ contains the identity function, $\mathcal{O}$ contains equality, and $\mathcal{C}$ includes the all values that can be computed by any sub-expression within the DSL G.

Notation: Given predicates $\mathcal{P} \subseteq \mathcal{U}$ and an abstract value φ, $\alpha^{\mathcal{P}}(\varphi)$ denotes the strongest conjunction of predicates in $\mathcal{P}$, such that, $\varphi \implies \alpha^{\mathcal{P}}(\varphi)$. Given a vector of abstract values $\boldsymbol{\varphi} = \langle \varphi_1, \ldots \varphi_n \rangle$, $\alpha^{\mathcal{P}}(\boldsymbol{\varphi})$ denotes the vector $\langle \alpha^{\mathcal{P}}(\varphi_1), \ldots \alpha^{\mathcal{P}}(\varphi_n) \rangle$. As is standard in the abstract interpretation literature [9], we use the notation $\gamma(\varphi)$ to

denote the set of concrete values represented by the abstract value φ.

Abstract semantics: In addition to the concrete semantics for each DSL construct, I assume we are given the abstract semantics for each function in the form of symbolic post-conditions over the universe of predicates $\mathcal{U}$. Given a production $s \to f(s_1, \ldots, s_n)$, $[\![f(\varphi_1, \ldots \varphi_k)]\!]^\#$ represents the abstract semantics of a function f. $[\![f(\varphi_1, \ldots, \varphi_k)]\!]^\# = \varphi$ if the function f returns φ (for symbol s), given abstract values $\varphi_1, \ldots, \varphi_k$ for arguments $s_1, \ldots, s_k$. I assume these abstract semantics are **sound**, i.e.,

$$[\![f(\varphi_1, \ldots, \varphi_k)]\!]^\# = \varphi \text{ and } v_1 \in \gamma(\varphi_1), \ldots v_k \in \gamma(\varphi_k) \implies [\![f(v_1, \ldots v_k)]\!] \in \gamma(\varphi)$$

However, I do not require the abstract semantics to be **precise**, i.e., formally, $[\![f(\varphi_1, \ldots, \varphi_k)]\!]^\# = \varphi$, there may exist a $v \in \varphi$, s.t.,

$$\forall\ v_1 \in \varphi_1, \ldots v_k \in \varphi_k . [\![f(v_1, \ldots v_k)]\!] \neq v$$

There may exist a concrete value v in the abstract output φ, such that, no concrete input parameters $v_1, \ldots v_k$ in the abstract inputs $\varphi_1, \ldots \varphi_k$ exist, for which $f(v_1, \ldots v_k) = v$.

I require the abstract semantics to be precise if all of the input parameters are abstract values representing a single concrete value, i.e., they are abstract values of the form $s = v$. Formally:

$$[\![f(s_1 = v_1, \ldots s_k = v_k)]\!]^\# = (s = [\![f(v_1, \ldots v_k)]\!])$$

Given a program p, predicates $\mathcal{P}$, and input x_j, $[\![p]\!]^{\mathcal{P}} x_j$ denotes the abstract value of program p, if the intermediate computed values are only represented via predicates in $\mathcal{P}$. Figure 6-1 presents the precise rules for computing $[\![p]\!]^{\mathcal{P}} x_j$. Given inputs $\boldsymbol{x} = \langle x_1, \ldots x_n \rangle$, $[\![p]\!]^{\mathcal{P}} \boldsymbol{x}$ denotes the vector $\boldsymbol{\varphi} = \langle [\![p]\!]^{\mathcal{P}} x_1, \ldots [\![p]\!]^{\mathcal{P}} x_n \rangle$.

Abstract Loss Function and Regularizer: I assume we are given the abstract semantics for the loss function and the regularizer. The abstract semantics of a loss

$$\frac{t \in T_C}{[\![t]\!]^{\mathcal{P}} x_j \Rightarrow \alpha^{\mathcal{P}}(t = [\![t]\!] x_j)} \text{ (CONSTANT)} \qquad \frac{}{[\![x]\!]^{\mathcal{P}} x_j \Rightarrow \alpha^{\mathcal{P}}(x = x_j)} \text{ (VARIABLE)}$$

$$\frac{[\![e_1]\!]^{\mathcal{P}} x_j \Rightarrow \varphi_1 \qquad [\![e_2]\!]^{\mathcal{P}} x_j \Rightarrow \varphi_2 \qquad \ldots \qquad [\![e_k]\!]^{\mathcal{P}} x_j \Rightarrow \varphi_k}{[\![f(e_1, e_2, \ldots e_k)]\!]^{\mathcal{P}} x_j \Rightarrow \alpha^{\mathcal{P}}([\![f(\varphi_1, \varphi_2, \ldots \varphi_k)]\!]^{\#})} \text{ (FUNCTION)}$$

Figure 6-1: Abstract execution semantics for program p.

function allows us to find the minimum possible loss value for a given abstract value. Formally, given a loss function $\mathcal{L}$, noisy output y, and an abstract value φ:

$$\mathcal{L}(\varphi, y) = \min_{z \in \gamma(\varphi)} \mathcal{L}(z, y)$$

Similarly, the abstract semantics of a regularizer allows us to find the minimum possible value of the regularizer, given an abstract value. Formally, given a regularizer $\mathcal{R}$, input x, and an abstract value φ:

$$\mathcal{R}(x, \varphi) = \min_{z \in \gamma(\varphi)} \mathcal{R}(x, z)$$

I assume that loss function $\mathcal{L}$ and regularizer $\mathcal{R}$ satisfies the following constraints:

$$\mathcal{L}(\langle z_1, \ldots z_n \rangle, \langle y_1, \ldots y_n \rangle) = \sum_{i=1}^{n} \mathcal{L}(z_i, y_i)$$

$$\mathcal{R}(\langle x_1, \ldots x_n \rangle, \langle z_1, \ldots z_n \rangle) = \sum_{i=1}^{n} \mathcal{R}(x_i, z_i)$$

Objective Function: Given a noisy dataset $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$, loss function $\mathcal{L}$, regularizer $\mathcal{R}$, and complexity measure C, we assume that the objective function U satisfies the following constraint:

$$\forall \delta \geq 0, \ \exists \delta_o \geq 0, \delta_r \geq 0$$

$$U(\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}]), C(p)) \leq \delta \implies \mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}_c) \leq \delta_o \text{ and } \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}]) \leq \delta_r$$

i.e., for any finite objective function value, both loss and regularizer weight are finite.

Abstract Objective Function: Given a predicates $\mathcal{P}$, noisy dataset $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$, objective function U, loss function $\mathcal{L}$, regularizer $\mathcal{R}$, complexity measure C, and a program p, I use an *abstract objective function* or *abstract objective function* value to denote the objective function value of a program p, if the abstract value $[\![p]\!]^{\mathcal{P}}\boldsymbol{x}$ is used as the noise-free output vector, i.e.,

$$U(\mathcal{L}([\![p]\!]^{\mathcal{P}}\boldsymbol{x}, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, [\![p]\!]^{\mathcal{P}}\boldsymbol{x}), C(p))$$

ϵ-correctness: In Chapter 3, I maintained a very strict version of correctness, i.e., the goal of the synthesis algorithm was to synthesize a program p^* which minimizes the objective function. This leaves out any speedups which can be achieved to by relaxing the requirement to synthesizing a program which is *close* to the optimal program but may not be one of the optimal programs.

Definition 18. (ϵ-correctness) *Given a noisy dataset $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$, a DSL G, an objective function U, a loss function $\mathcal{L}$, a regularizer $\mathcal{R}$, and a complexity measure C, a program $p_r \in G$ is* **ϵ-correct***, if and only if,*

$$U(\mathcal{L}(p_r[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p_r[\boldsymbol{x}]), C(p_r)) - \min_{p \in G} U(\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}]), C(p)) \leq \epsilon$$

A program $p_r \in G$ is **ϵ-correct**, if and only if, its objective function value is at most ϵ greater than the minimum possible objective function value for any program in G. Note that, for $\epsilon = 0$, $p_r \in \mathsf{argmin}_{p \in G} U(\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}]), C(p))$. For my experiments, ϵ is set to 0.

6.2 Abstract Finite Tree Automaton

An abstract finite tree automaton (AFTA) is a generalized version of a concrete finite tree automaton. It replaces concrete values in a CFTA with abstract values. Using abstract values allows us to represent multiple concrete value states with a single abstract value state. This allows the synthesis algorithm to compress the size of the automaton.

$$\frac{\boldsymbol{\varphi} = \alpha^{\mathcal{P}}\Big(\langle x = x_1, \ldots x = x_n\rangle\Big)}{q_x^{\boldsymbol{\varphi}} \in Q}\ (\textsc{Var}) \quad \frac{q_{s_0}^{\boldsymbol{\varphi}} \in Q}{q_{s_0}^{\boldsymbol{\varphi}} \in Q_f}\ (\textsc{Final})$$

$$\frac{t \in T_C, \boldsymbol{\varphi} = \alpha^{\mathcal{P}}\Big(\langle t = [\![t]\!], \ldots t = [\![t]\!]\rangle\Big), |\boldsymbol{\varphi}| = n}{q_t^{\boldsymbol{\varphi}} \in Q}\ (\textsc{Const})$$

$$\frac{\begin{array}{c} s \to f(s_1, \ldots, s_k) \in P, q_{s_1}^{\boldsymbol{\varphi}_1}, \ldots, q_{s_k}^{\boldsymbol{\varphi}_k} \in Q, \\ \varphi_j = \alpha^{\mathcal{P}}\Big([\![f(\boldsymbol{\varphi}_{1j}, \ldots, \boldsymbol{\varphi}_{kj})]\!]^{\#}\Big), \boldsymbol{\varphi} = \langle \varphi_1, \ldots \varphi_n\rangle \end{array}}{q_s^{\boldsymbol{\varphi}} \in Q, f(q_{s_1}^{\boldsymbol{\varphi}_1}, \ldots, q_{s_k}^{\boldsymbol{\varphi}_k}) \to q_s^{\boldsymbol{\varphi}} \in \Delta}\ (\textsc{Prod})$$

Figure 6-2: Rules for constructing FTA $\mathcal{A} = (Q, F, Q_f, \Delta)$ with abstract values, for inputs $\boldsymbol{x} = \langle x_1, \ldots x_n\rangle$.

Given predicates $\mathcal{P}$, DSL G, and inputs $\boldsymbol{x}$, Figure 6-2 presents the rules for constructing an AFTA (Q, Q_f, Δ). States in an AFTA are of the form $q_s^{\boldsymbol{\varphi}}$, where s is a symbol and $\boldsymbol{\varphi}$ is a vector of abstract values. If $q_s^{\boldsymbol{\varphi}} \in Q$, then there exists an expression e, starting from symbol s, such that, $[\![e]\!]^{\mathcal{P}}\boldsymbol{x} = \boldsymbol{\varphi}$, i.e., there exists a subexpression in G starting from symbol s, such that, the abstract value of that expression is $\boldsymbol{\varphi}$, given inputs $\boldsymbol{x}$. If there is a transition $f(q_{s_1}^{\boldsymbol{\varphi}_1}, \ldots q_{s_k}^{\boldsymbol{\varphi}_k}) \to q_s^{\boldsymbol{\varphi}}$ in the AFTA, then

$$\forall j \in [1, |\boldsymbol{\varphi}|].[\![f(\varphi_{1j}, \ldots \varphi_{kj})]\!]^{\#} \implies \varphi_j$$

The Var Rule constructs a state $q_x^{\boldsymbol{\varphi}} \in Q$ for the variable symbol x, where $\boldsymbol{\varphi} = \alpha^{\mathcal{P}}(\langle x = x_1, \ldots x = x_n\rangle)$. The Const Rule constructs a state $q_t^{\boldsymbol{\varphi}} \in Q$ for each constant terminal t, where $\boldsymbol{\varphi}$ is a vector of size equal to $\boldsymbol{x}$ with each entry as $\alpha^{\mathcal{P}}(t = [\![t]\!])$. The Final Rule adds all states with symbol s_0 are added to Q_f, where s_0 is the start symbol. The Prod Rule constructs a state $q_s^{\boldsymbol{\varphi}} \in Q$, if there exists a production $s \to f(s_1, \ldots s_n) \in P$ and there exists states $q_{s_1}^{\boldsymbol{\varphi}_1}, \ldots q_{s_k}^{\boldsymbol{\varphi}_k} \in Q$, where $\forall i = [1, |\boldsymbol{\phi}|].f(\varphi_{1i}, \ldots \varphi_{ki}) \implies \varphi_i$. The Prod Rule also adds a transition $f(q_{s_1}^{\boldsymbol{\varphi}_1}, \ldots q_{s_k}^{\boldsymbol{\varphi}_k}) \to q_s^{\boldsymbol{\varphi}}$ in Δ, if it constructs a such a state.

Theorem 21. (Structure of the Tree Automaton) *Given a set of predicates $\mathcal{P}$, input vector $\boldsymbol{x} = \langle x_1, \ldots x_n\rangle$, and DSL G, let $\mathcal{A} = (Q, Q_f, \Delta)$ be the AFTA returned by the function* $\mathsf{ConstructAFTA}(\boldsymbol{x}, G, \mathcal{P})$. *Then for all symbols s in G, for all expressions*

e starting from symbol s (and height less than bound b), there exists a state $q_s^{\boldsymbol{\varphi}} \in Q$*, such that, e is accepted by the automaton* $(Q, \{q_s^{\boldsymbol{\varphi}}\}, \Delta)$*, where* $\boldsymbol{\varphi} = \langle [\![e]\!]^{\mathcal{P}} x_1, \ldots [\![e]\!]^{\mathcal{P}} x_n \rangle$*.*

Proof. I prove this theorem by using induction over height of the expression e.

Base Case: Height of expression e is 1. This implies the symbol is either x or a constant. According to Var and Const rules (Figure 6-2), there exists state $q_t^{\boldsymbol{\varphi}} \in Q$ (for terminal t), where $\boldsymbol{\varphi} = \langle [\![t]\!]^{\mathcal{P}} x_1, \ldots [\![t]\!]^{\mathcal{P}} x_n \rangle$ and t is accepted by automaton $(Q, \{q_t^{\boldsymbol{\varphi}}\}, \Delta)$.

Inductive Hypothesis: For all symbols s in G, for all expressions e starting from symbol s of height less than equal to n, there exists a state $q_s^{\boldsymbol{\varphi}} \in Q$, such that, e is accepted by the automaton $(Q, \{q_s^{\boldsymbol{\varphi}}\}, \Delta)$, where $\boldsymbol{\varphi} = \langle [\![e]\!]^{\mathcal{P}} x_1, \ldots [\![e]\!]^{\mathcal{P}} x_n \rangle$.

Induction Step: For any symbol s in G, consider an expression $e = f(e_1, \ldots e_k)$ of height equal to $n+1$, created from production $s \leftarrow f(s_1, \ldots s_k)$. Note the height of expressions $e_1, \ldots e_k$ is less than equal to n, therefore using induction hypothesis, there exists states $q_{s_1}^{\boldsymbol{\varphi}_1}, \ldots q_{s_k}^{\boldsymbol{\varphi}_k} \in Q$, such that, e_i is accepted by automaton $(Q, \{q_{s_i}^{\boldsymbol{\varphi}_i}\}, \Delta)$, where $\boldsymbol{\varphi}_i = \langle [\![e_i]\!]^{\mathcal{P}} x_1, \ldots [\![e_i]\!]^{\mathcal{P}} x_n \rangle$.

Note based on abstract execution rules (Figure 6-1):

$$[\![e]\!]^{\mathcal{P}} x_i = \alpha^{\mathcal{P}}([\![f([\![e_1]\!]^{\mathcal{P}} x_i, \ldots [\![e_k]\!]^{\mathcal{P}} x_i)]\!]^{\#})$$

According to Prod rule (Figure 6-2), there exists a state $q_s^{\boldsymbol{\varphi}} \in Q$, where

$$\boldsymbol{\varphi} = \langle [\![e]\!]^{\mathcal{P}} x_1, \ldots [\![e]\!]^{\mathcal{P}} x_n \rangle$$

and e is accepted by $(Q, \{q_s^{\boldsymbol{\varphi}}\}, \Delta)$.

Therefore, by induction, for all symbols s in G, for all expressions e starting from symbol s (and height less than bound b), there exists a state $q_s^{\boldsymbol{\varphi}} \in Q$, such that, e is accepted by the automaton $(Q, \{q_s^{\boldsymbol{\varphi}}\}, \Delta)$, where $\boldsymbol{\varphi} = \langle [\![e]\!]^{\mathcal{P}} x_1, \ldots [\![e]\!]^{\mathcal{P}} x_n \rangle$. □

Corollary 1. *Given a set of predicates* $\mathcal{P}$*, input vector* $\boldsymbol{x} = \langle x_1, \ldots x_n \rangle$*, and DSL* G*, let* $\mathcal{A} = (Q, Q_f, \Delta)$ *be the AFTA returned by the function* $\mathsf{ConstructAFTA}(\boldsymbol{x}, G, \mathcal{P})$*. All programs p (of height less than bound b) are accepted by* $\mathcal{A}$*. For any accepting*

state $q^{\varphi}_{s_0} \in Q_f$*, a program* p *is accepted by the automaton* $(Q, \{q^{\varphi}_{s_0}\}, \Delta)$*, if and only if,* $\forall i \in [1, n]$*,* $p[x_i] \in \gamma(\varphi_i)$*.*

These theorems state that 1) all programs are accepted by some accepting state in the AFTA 2) if a program is accepted by an accepting state then its abstract value is equal to the abstract value attached to the accepting state.

6.3 Synthesis Algorithm

Figure 6-3 presents my synthesis algorithm. Given a noisy dataset $\mathcal{D}$, a DSL G, a tolerance level $\epsilon \geq 0$, initial predicates $\mathcal{P}$, a universe of possible predicates $\mathcal{U}$, objective function U, loss function $\mathcal{L}$, regularizer $\mathcal{R}$, and a complexity measure C, Synthesize returns a program p^* which satisfies the ϵ-correctness condition (Definition 18). I assume that $\mathsf{true}, \mathsf{false} \in \mathcal{P}$. All the procedures and sub-procedures are parameterized by an objective function, a loss function, a regularizer, and a complexity measure. I remove these parameters from the signature of Synthesize (and other methods) for simplicity.

The synthesis algorithm consists of a refinement loop (line 2-7). The loop first constructs an abstract finite tree automaton (line 3) with the current set of predicates $\mathcal{P}$ using rules presented in Figure 6-2. The algorithm then uses the MinCost function to generate a candidate program p (line 4). Given predicates $\mathcal{P}$, this candidate program minimizes the *abstract objective function*.

If the *distance* between the current program p and the best possible program in the DSL G, is less a tolerance level ϵ (Distance function, line 5), the algorithm returns the candidate program p. Otherwise, the algorithm refines the AFTA to either improve the estimation of the best possible program or synthesize a better candidate program. To refine the AFTA, the algorithm first picks an input-output example (x, y) from dataset $\mathcal{D}$, on which I can improve the candidate program p (line 6). Given an input-output example (x, y), the procedure OptimizeAndBackPropagate constructs the constraints required to improve the AFTA and then returns the set of predicates which will allow the algorithm to build a more refined AFTA.

I discuss each of these sub-procedures in detail next.

1: **procedure** SYNTHESIZE($\mathcal{D}, G, \epsilon, \mathcal{P}, \mathcal{U}$)
input: noisy dataset $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$, DSL G, and tolerance ϵ.
input: initial predicates $\mathcal{P}$, and universe of predicates $\mathcal{U}$.
output: A program p, such that, p satisfies the ϵ-*correctness* condition.
2: **while** true **do**
3: $\mathcal{A} :=$ ConstructAFTA$(\boldsymbol{x}, G, P)$;
4: $p :=$ MinCost$(\mathcal{A}, \mathcal{D})$;
5: **if** Distance$(p, \mathcal{D}, \mathcal{P}) \leq \epsilon$ **then return** p;
6: $x, y :=$ PickDimension$(p, \mathcal{D}, \mathcal{P})$;
7: $\mathcal{P} := \mathcal{P} \bigcup$ OptimizeAndBackPropogate$(p, x, y, \mathcal{P}, \mathcal{U})$;

Figure 6-3: Algorithm for noisy program synthesis using abstraction refinement based optimization.

1: **procedure** MINCOST($\mathcal{A}, \mathcal{D}$)
input: AFTA $\mathcal{A} = (Q, Q_f, \Delta)$, noisy dataset $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$.
output: Program $p^* \in \arg\min_{p \in G} U(\mathcal{L}([\![p]\!]^{\mathcal{P}}\boldsymbol{x}, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, [\![p]\!]^{\mathcal{P}}\boldsymbol{x}), C(p))$
2: $p^* :=$ null; $u^* := \infty$;
3: **for** $q^{\boldsymbol{\varphi}}_{s_0} \in Q_f$ **do**
4: $p, c :=$ LeastComplex$(q^{\boldsymbol{\varphi}}_{s_0}, \mathcal{A}, G)$
5: $u := U(\mathcal{L}(\boldsymbol{\varphi}, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, \boldsymbol{\varphi}), C(p))$;
6: **if** $u \leq u^*$ **then**
7: $p^* := p$; $u^* := u$;
8: **return** p^*;

Figure 6-4: Procedure for synthesizing the program which minimizes the abstract objective function.

6.4 Minimum Cost Candidate

I present the implementation of procedure MinCost in Figure 6-4. Given an AFTA (Q, Q_f, Δ), noisy dataset $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$, objective function U, loss function $\mathcal{L}$, regularizer $\mathcal{R}$, and complexity measure C, MinCost returns a program p^*, which minimizes the *abstract* objective function, i.e., for all programs $p \in G$:

$$U(\mathcal{L}([\![p^*]\!]^{\mathcal{P}}\boldsymbol{x}, \boldsymbol{y}), \mathcal{R}([\![p^*]\!]^{\mathcal{P}}\boldsymbol{x}, \boldsymbol{x}), C(p^*)) \leq U(\mathcal{L}([\![p]\!]^{\mathcal{P}}\boldsymbol{x}, \boldsymbol{y}), \mathcal{R}([\![p]\!]^{\mathcal{P}}\boldsymbol{x}, \boldsymbol{x}), C(p))$$

Note that, since for all programs p, $\mathcal{L}([\![p]\!]^{\mathcal{P}}\boldsymbol{x}, \boldsymbol{y}) \leq \mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y})$ and $\mathcal{R}([\![p]\!]^{\mathcal{P}}\boldsymbol{x}, \boldsymbol{x}) \leq \mathcal{R}(p[\boldsymbol{x}], \boldsymbol{x})$, the following statement is true:

$$U(\mathcal{L}([\![p]\!]^{\mathcal{P}}\boldsymbol{x}, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, [\![p]\!]^{\mathcal{P}}\boldsymbol{x}), C(p)) \leq U(\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}]), C(p))$$

Hence, for programs $p \in G$:

$$U(\mathcal{L}([\![p^*]\!]^{\mathcal{P}}\boldsymbol{x}, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, [\![p^*]\!]\boldsymbol{x}), C(p^*)) \leq U(\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}]), C(p))$$

The procedure iterates through all accepting states $q^{\boldsymbol{\varphi}}_{s_0} \in Q_f$, to find the program p^* which minimizes the *abstract* objective function. For a given accepting state $q^{\boldsymbol{\varphi}}_{s_0}$, the procedure finds the least complex program p (i.e., program which minimizes the complexity measure) which is accepted by $(Q, \{q^{\boldsymbol{\varphi}}_{s_0}\}, \Delta)$ (line 4). Given an accepting state of the form $q^{\boldsymbol{\varphi}}_{s_0}$, a program $p \in G$ is accepted by the automaton $(Q, \{q^{\boldsymbol{\varphi}}_{s_0}\}, \Delta)$, if and only if, $[\![p]\!]^{\mathcal{P}}\boldsymbol{x} = \boldsymbol{\varphi}$. Given an accepting $q^{\boldsymbol{\varphi}}_{s_0}$, p (line 4) is the least complex program which maps input vector $\boldsymbol{x}$ to abstract outputs $\boldsymbol{\varphi}$, i.e.,

$$p \in \mathsf{argmin}_{p \in G[\boldsymbol{x} \to \boldsymbol{\varphi}]} C(p)$$

where $G[\boldsymbol{x} \to \boldsymbol{\varphi}] = \{p \mid p \in G, [\![p]\!]^{\mathcal{P}}\boldsymbol{x} = \boldsymbol{\varphi}\}$.

Given state $q^{\boldsymbol{\varphi}}_{s_0}$, the algorithm then computes the *abstract* objective function value for p, (the least complex program) i.e., $U(\mathcal{L}(\boldsymbol{\varphi}, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, \boldsymbol{\varphi}), C(p))$. The algorithm iterates through all states and returns the program which minimizes the *abstract* objective function.

Theorem 22. *Given predicates $\mathcal{P}$, DSL G, dataset $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$, objective function U, loss function $\mathcal{L}$, regularizer $\mathcal{R}$, complexity measure C, and $\mathcal{A} = \mathsf{ConstructAFTA}(\boldsymbol{x}, G, \mathcal{P})$, if $p^* = \mathsf{MinCost}(\mathcal{A}, \mathcal{D})$, then*

$$p^* \in \mathsf{argmin}_{p \in G} U(\mathcal{L}([\![p]\!]^{\mathcal{P}}\boldsymbol{x}, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, [\![p]\!]^{\mathcal{P}}\boldsymbol{x}), C(p))$$

i.e., p^* minimizes the abstract objective function.

Proof. Let $\mathcal{A} = (Q, Q_f, \Delta)$. From Corollary 1, for each program $p \in G$, there exists a state $q^{\boldsymbol{\varphi}}_{s_0} \in Q_f$, such that, $[\![p]\!]^{\mathcal{P}}\boldsymbol{x} = \boldsymbol{\varphi}$. Let $P[q^{\boldsymbol{\varphi}}_{s_0}], c = \mathsf{LeastComplex}(q^{\boldsymbol{\varphi}}_{s_0}, \mathcal{A}, G)$. The algorithm finds an accepting state $q^{\boldsymbol{\varphi}^*}_{s_0} \in Q_f$, such that, for all accepting states

$q_{s_0}^{\boldsymbol{\varphi}} \in Q_f$,

$$U(\mathcal{L}(\boldsymbol{\varphi}^*, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, \boldsymbol{\varphi}^*), C(P[q_{s_0}^{\boldsymbol{\varphi}^*}])) \leq U(\mathcal{L}(\boldsymbol{\varphi}, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, \boldsymbol{\varphi}), C(P[q_{s_0}^{\boldsymbol{\varphi}}]))$$

for all $p \in G$,

$$U(\mathcal{L}(\boldsymbol{\varphi}^*, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, \boldsymbol{\varphi}^*), C(P[q_{s_0}^{\boldsymbol{\varphi}^*}])) \leq U(\mathcal{L}([\![p]\!]^{\mathcal{P}}\boldsymbol{x}, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, [\![p]\!]^{\mathcal{P}}\boldsymbol{x}), C(p))$$

Since $p^* = P[q_{s_0}^{\boldsymbol{\varphi}^*}]$,

$$p^* \in \mathsf{argmin}_{p \in G} U(\mathcal{L}([\![p]\!]^{\mathcal{P}}\boldsymbol{x}, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, [\![p]\!]^{\mathcal{P}}\boldsymbol{x}), C(p))$$

□

6.5 Termination Condition and Tolerance

Given a candidate program p^*, predicates $\mathcal{P}$, noisy dataset $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$, objective function U, loss function L, regularizer $\mathcal{R}$, and complexity measure C, the Distance function returns the difference between the concrete objective function value of program p^* over noisy dataset $\mathcal{D}$ and the abstract objective function value (given predicates from $\mathcal{P}$) over noisy dataset $\mathcal{D}$. Formally:

$$\mathsf{Distance}\Big(p, (\boldsymbol{x}, \boldsymbol{y}), \mathcal{P}\Big) :=$$

$$U(\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}]), C(p)) - U(\mathcal{L}([\![p]\!]^{\mathcal{P}}\boldsymbol{x}, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, [\![p]\!]^{\mathcal{P}}\boldsymbol{x}), C(p))$$

The algorithm terminates if the distance is less than equal to the tolerance level ϵ, i.e.,

$$U(\mathcal{L}(p^*[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p^*[\boldsymbol{x}]), C(p^*)) - U(\mathcal{L}([\![p^*]\!]^{\mathcal{P}}\boldsymbol{x}, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, [\![p^*]\!]^{\mathcal{P}}\boldsymbol{x}), C(p^*)) \leq \epsilon$$

Since, for all programs $p \in G$:

$$U(\mathcal{L}([\![p^*]\!]^{\mathcal{P}}\boldsymbol{x}, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, [\![p^*]\!]^{\mathcal{P}}\boldsymbol{x}), C(p^*)) \leq U(\mathcal{L}([\![p]\!]^{\mathcal{P}}\boldsymbol{x}, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, [\![p]\!]^{\mathcal{P}}\boldsymbol{x}), C(p))$$

$$U(\mathcal{L}([\![p]\!]^{\mathcal{P}}\boldsymbol{x}, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, [\![p]\!]^{\mathcal{P}}\boldsymbol{x}), C(p)) \leq U(\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}]), C(p))$$

the following statement is also true:

$$U(\mathcal{L}(p^*[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p^*[\boldsymbol{x}]), C(p^*)) - \min_{p \in G} U(\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}]), C(p)) \leq \epsilon$$

Theorem 23. (Soundness) *Given a dataset $\mathcal{D}$, a DSL G, tolerance $\epsilon \geq 0$, universe of predicates $\mathcal{U}$, initial predicates $\mathcal{P}$, objective function U, loss function $\mathcal{L}$, regularizer $\mathcal{R}$, and the complexity measure C, if Algorithm 6-3 returns the program p^*, then p^* satisfies the ϵ-correctness condition (Definition 18).*

Proof. Let us assume that the algorithm terminates on the i^{th} iteration. Let $\mathcal{A}_i = (Q, Q_f, \Delta)$ be the AFTA when the algorithm terminates. Let p_i be the program returned by MinCost on the i^{th} iteration.

From Theorem 22, for all programs $p \in G$,

$$U(\mathcal{L}([\![p_i]\!]^{\mathcal{P}}\boldsymbol{x}, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, [\![p_i]\!]^{\mathcal{P}}\boldsymbol{x}), C(p_i)) \leq U(\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}]), C(p))$$

When the algorithm terminates, the following condition is true:

$$\mathsf{Distance}(p_i, \mathcal{D}, \mathcal{P}) \leq \epsilon$$

which implies:

$$U(\mathcal{L}(p_i[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p_i[\boldsymbol{x}]), C(p_i)) - U(\mathcal{L}([\![p_i]\!]^{\mathcal{P}}\boldsymbol{x}, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, [\![p_i]\!]^{\mathcal{P}}\boldsymbol{x}), C(p_i)) \leq \epsilon$$

Therefore, for all programs $p \in G$,

$$U(\mathcal{L}(p_r[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p_r[\boldsymbol{x}]), C(p_r)) - U(\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}]), C(p)) \leq \epsilon$$

Hence, if the Algorithm 6-3 returns a program p_i, then p_i satisfies the *ϵ-correctness* condition. □

6.6 Abstraction Refinement Based Optimization

Given a dataset $\mathcal{D}$ and predicates $\mathcal{P}$, the program p^* (returned by MinCost) minimizes the *abstract* objective function, i.e., for all programs $p \in G$:

$$U(\mathcal{L}([\![p^*]\!]^{\mathcal{P}}\boldsymbol{x}, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, [\![p^*]\!]^{\mathcal{P}}\boldsymbol{x}), C(p^*)) \leq U(\mathcal{L}([\![p]\!]^{\mathcal{P}}\boldsymbol{x}, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, [\![p]\!]^{\mathcal{P}}\boldsymbol{x}), C(p))$$

Since the algorithm did not terminate, $\mathsf{Distance}(p^*, \mathcal{P}, \mathcal{D}) > \epsilon$.

Let us consider the case when $\epsilon = 0$. Since $\mathsf{Distance}(p^*, \mathcal{P}, \mathcal{D}) > 0$, the concrete objective function value of program p^*, over dataset $\mathcal{D}$, is greater than the abstract objective function value of p^* over $\mathcal{D}$. Formally,

$$U(\mathcal{L}([\![p^*]\!]^{\mathcal{P}}\boldsymbol{x}, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, [\![p^*]\!]^{\mathcal{P}}\boldsymbol{x}), C(p^*)) < U(\mathcal{L}(p^*[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p^*[\boldsymbol{x}]), C(p^*))$$

At this point, even though, for all programs $p \in G$, the following is true:

$$U(\mathcal{L}([\![p^*]\!]^{\mathcal{P}}\boldsymbol{x}, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, [\![p^*]\!]^{\mathcal{P}}\boldsymbol{x}), C(p^*)) < U(\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}]), C(p))$$

We cannot prove that p^* is the optimal function, i.e.,

$$U(\mathcal{L}(p^*[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p^*[\boldsymbol{x}]), C(p^*)) < U(\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}]), C(p))$$

And therefore, just using predicates $\mathcal{P}$, we cannot prove that p^* is the optimal program. Similarly, if $\epsilon > 0$, for all programs $p \in G$, the following is true:

$$U(\mathcal{L}([\![p^*]\!]^{\mathcal{P}}\boldsymbol{x}, \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, [\![p^*]\!]^{\mathcal{P}}\boldsymbol{x}), C(p^*)) < U(\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}]), C(p))$$

But we cannot prove that the following statement is true:

$$U(\mathcal{L}(p^*[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p^*[\boldsymbol{x}]), C(p^*)) - \min_{p \in G} U(\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}]), C(p)) \leq \epsilon$$

And therefore, just using predicates $\mathcal{P}$, we cannot prove that p_r is ϵ-correct. Therefore, in order to find the optimal program and prove its optimality, we have to expand

the set of predicates $\mathcal{P}$.

To achieve this goal, the algorithm first selects an input-output example from the noisy dataset $\mathcal{D}$ on which the algorithm can improve the difference between the abstract objective function value and the concrete objective function value of programs using procedure PickDimension. Given an input-output example (x, y) returned by PickDimension, the idea here is to expand the set of predicates $\mathcal{P}$ to $\mathcal{P}'$, such that:

$$\mathcal{L}([\![p^*]\!]^{\mathcal{P}} x, y) < \mathcal{L}([\![p^*]\!]^{\mathcal{P}'} x, y) \leq \mathcal{L}(p^*[x], y)$$

or

$$\mathcal{R}(x, [\![p^*]\!]^{\mathcal{P}} x) < \mathcal{R}(x, [\![p^*]\!]^{\mathcal{P}'} x) \leq \mathcal{L}(x, p^*[x])$$

Then the algorithm rebuilds the abstract finite state automaton using these expanded set of predicates. Note that, for any program p,

$$\forall x \in \boldsymbol{x}.\ [\![p]\!]^{\mathcal{P}} x \implies [\![p]\!]^{\mathcal{P}'} x \implies (s_0 = p[x])$$

but $[\![p]\!]^{\mathcal{P}} \boldsymbol{x}$ may not be equal to $[\![p]\!]^{\mathcal{P}'} \boldsymbol{x}$. Therefore, the AFTA built on the expanded set of predicates $\mathcal{P}'$ may have more states compared to AFTA built on predicates $\mathcal{P}$. The refined predicates improves the estimation of the abstract loss function or abstract regularizer for the candidate program (and potentially other programs).

The algorithm allows us to plug any implementation of the procedure PickDimension, assuming it satisfies the following constraint:

$$\langle x_i, y_i \rangle = \mathsf{PickDimension}\Big(p, (\boldsymbol{x}, \boldsymbol{y}), \mathcal{P}, L\Big) \implies$$
$$\mathcal{L}(p[x_i], y_i) > \mathcal{L}([\![p]\!]^{\mathcal{P}} x_i, y_i) \text{ or } \mathcal{R}(x_i, p[x_i]) > \mathcal{R}(x_i, [\![p]\!]^{\mathcal{P}} x_i)$$

Since $\mathsf{Distance}(p^*, \mathcal{D}, \mathcal{P}) > \epsilon$, there exists at least one $i \in [1, n]$, such that,

$$\mathcal{L}(p^*[x_i], y_i) > \mathcal{L}([\![p^*]\!]^{\mathcal{P}} x_i, y_i) \text{ or } \mathcal{R}(x_i, p^*[x_i]) > \mathcal{R}(x_i, [\![p^*]\!]^{\mathcal{P}} x_i)$$

If multiple input-output examples exist for which the above condition holds, an imple-

1: **procedure** OPTIMIZEANDBACKPROPAGATE($p, x_j, y_j, \mathcal{P}, \mathcal{U}$)
input: Program p, input x_j, noisy output y_j, predicates $\mathcal{P}$, and universe of predicates $\mathcal{U}$.
output: A set of predicates $\mathcal{P}_r$.
2: $\varphi := [\![p]\!]^{\mathcal{P}} x_j$; $\phi := (s_0 = [\![p]\!] x_j)$;
3: $\Phi := \left\{ q \in \mathcal{U} | \phi \implies q \right\}$;
4: $\Psi := \Phi$;
5: **for** $i = 1 \ldots m$ **do** ▷ Use a maximum of m predictates.
6: $\Psi := \Psi \bigcup \left\{ \psi \wedge q \mid \psi \in \Psi, q \in \Phi \right\}$;
7: $\psi^* := \phi$;
8: **for** $\psi \in \Psi$ **do**
9: **if** $\psi^* \implies \psi$ **then**
10: **if** $\mathcal{L}(\varphi, y_j) - \mathcal{L}(\varphi \wedge \psi, y_j) \geq \delta$ or $\mathcal{R}(x_j, \varphi) - \mathcal{R}(x_j, \varphi \wedge \psi) \geq \delta$ **then**
11: $\psi^* := \psi$;
▷ The abstract loss is increased by atleast $\delta > 0$.
12: **return** ExtractPredicates$(\psi^*) \bigcup$ BackPropagate$(p, x_j, \varphi \wedge \psi^*, \mathcal{P}, \mathcal{U})$;

Figure 6-5: Algorithm for extracting predicates $\mathcal{P}_r$ to refine the abstract value of p, such that, $\mathcal{L}([\![p]\!]^{\mathcal{P}} x_j, y_j) < \mathcal{L}([\![p]\!]^{(\mathcal{P} \bigcup \mathcal{P}_r)} x_j, y_j)$ or $\mathcal{R}(x_j, [\![p]\!]^{\mathcal{P}} x_j) < \mathcal{R}(x_j, [\![p]\!]^{(\mathcal{P} \bigcup \mathcal{P}_r)} x_j)$.

mentation of PickDimension can return any one of them and my synthesis algorithm will use that example to optimize the automaton.

Given an example (x, y), the algorithm uses the procedure OptimizeAndBackPropagate to expand the set of predicates to $\mathcal{P}'$, such that

$$\mathcal{L}([\![p^*]\!]^{\mathcal{P}} x, y) < \mathcal{L}([\![p^*]\!]^{\mathcal{P}'} x, y) \leq \mathcal{L}(p^*[x], y)$$
$$\text{or } \mathcal{R}(x, [\![p^*]\!]^{\mathcal{P}} x) < \mathcal{R}(x, [\![p^*]\!]^{\mathcal{P}'} x) \leq \mathcal{R}(x, p^*[x])$$

Figure 6-5 presents the OptimizeAndBackPropagate procedure. The procedure synthesizes the strongest formula ψ^*, such that, $(s_0 = p^*[x]) \implies \psi^*$ and:

$$\mathcal{L}([\![p^*]\!]^{\mathcal{P}} x, y) < \mathcal{L}(([\![p^*]\!]^{\mathcal{P}} x) \wedge \psi^*, y) \text{ and } \mathcal{R}(x, [\![p^*]\!]^{\mathcal{P}} x) < \mathcal{R}(x, ([\![p^*]\!]^{\mathcal{P}} x) \wedge \psi^*)$$

Note that since $(s_0 = [\![p^*]\!] x) \implies \psi^*$ (line 7):

$$\mathcal{L}(([\![p^*]\!]^{\mathcal{P}} x) \wedge \psi^*, y) \leq \mathcal{L}(p^*[x], y) \text{ or } \mathcal{R}(x, ([\![p^*]\!]^{\mathcal{P}} x) \wedge \psi^*) \leq \mathcal{R}(x, p^*[x])$$

1: **procedure** BACKPROPOGATE($f(e_1, \ldots e_n), x_j, \psi_p, \mathcal{P}, \mathcal{U}$)
output: A set of predicates P_r, such that, $[\![f(e_1, \ldots e_n)]\!]^{(\mathcal{P} \cup \mathcal{P}_r)} x_j \implies \psi_p$.
2: $\boldsymbol{\phi} := \langle [\![e_1]\!] x_j, \ldots [\![e_n]\!] x_j \rangle$; $\boldsymbol{\varphi} := \langle [\![e_1]\!]^{\mathcal{P}} x_j, \ldots [\![e_n]\!]^{\mathcal{P}} x_j \rangle$;
3: $\boldsymbol{\Phi} := \langle \Phi_1, \ldots \Phi_n \rangle$ where $\Phi_i := \left\{ q \in \mathcal{U} | \phi_i \implies q \right\}$; $\boldsymbol{\Psi} := \boldsymbol{\Phi}$;
4: **for** $i = 1 \ldots m$ **do** ▷ Use a maximum of m predicates.
5: **for** $j = 1, \ldots, n$ **do**
6: $\Psi_j := \Psi_j \bigcup \left\{ \psi \wedge q \mid \psi \in \Psi_j, q \in \Phi_j \right\}$;
7: $\boldsymbol{\psi^*} := \boldsymbol{\phi}$;
8: **for all** $\boldsymbol{\psi} = \langle \psi_1, \ldots \psi_n \rangle \mid \psi_i \in \Psi_i$ **do**
9: **if** $\forall i = 1, \ldots n.\ \psi_i^* \implies \psi_i$ and $[\![f(\varphi_1 \wedge \psi_1, \ldots \varphi_n \wedge \psi_n)]\!]^{\#} \implies \psi_p$ **then**
10: $\boldsymbol{\psi^*} := \boldsymbol{\psi}$;
11: $\mathcal{P}_r := \emptyset$;
12: **for** $i = 1 \ldots n$ **do**
13: $\mathcal{P}_r := \mathcal{P}_r \bigcup \mathsf{ExtractPredicates}(\psi_i^*)$;
14: **if** $e_i \notin T$ **then** $\mathcal{P}_r := \mathcal{P}_r \bigcup \mathsf{BackPropogate}(e_i, x_j, \varphi_i \wedge \psi_i^*, \mathcal{P}, \mathcal{U})$;
15: **return** $\mathcal{P}_r$;

Figure 6-6: Algorithm to back propagate abstract value $\varphi \wedge \psi^*$ of expression $e = f(e_1, \ldots e_k)$, such that, $[\![f(\varphi_1 \wedge \psi_1^*, \ldots \varphi_k \wedge \psi_k^*)]\!]^{\#} \implies \psi_p$.

Theorem 24. *Given expression $e = f(e_1, \ldots e_n)$, input x, abstract value ψ_p (assuming $(s = [\![e]\!]x) \implies \psi_p$), predicates $\mathcal{P}$, and universe of predicates $\mathcal{U}$, if the procedure* $\mathsf{BackPropagate}(e, x, \psi_p, \mathcal{P}, \mathcal{U})$ *returns predicate set $\mathcal{P}_r$ then:*

$$[\![e]\!]^{\mathcal{P} \cup \mathcal{P}_r} x \implies \psi_p$$

Proof. I prove this theorem using induction over height of expression e.

Base Case: Height of e is 2. This means all sub-expressions $e_1, \ldots e_k$ are terminals. Note that $\mathcal{P}_r \subseteq \mathsf{ExtractPredicates}(\psi_i^*)$, for all $i \in [1, k]$.

$$[\![e_i]\!]^{\mathcal{P} \cup \mathcal{P}_r} \implies \varphi_i \wedge \psi_i^*$$

and

$$[\![f(\varphi_1 \wedge \psi_1^*, \ldots \varphi_k \wedge \psi_k^*)]\!]^{\#} \implies \psi_p$$

therefore

$$[\![e]\!]^{\mathcal{P} \cup \mathcal{P}_r} x \implies \psi_p$$

Induction Hypothesis: For all expressions e of height less than equal to n, the following is true:

$$[\![e]\!]^{\mathcal{P}\cup\mathcal{P}_r}x \implies \psi_p$$

Induction Step: Let $e = f(e_1, \ldots e_k)$ be an expression of height equal to $n+1$. The height of expressions $e_1, \ldots e_k$ is less than equal to n.

Note that $\varphi_i \wedge \psi_i^* \implies [\![e_i]\!]x$ (line-7 and line-9). And since $\mathsf{BackPropagate}(e_i, x, \varphi_i \wedge \psi_i^*, \mathcal{P}, \mathcal{U}) \subseteq \mathcal{P}_r$, using induction hypothesis:

$$[\![e_i]\!]^{\mathcal{P}\cup\mathcal{P}_r}x \implies \varphi_i \wedge \psi_i^*$$

and

$$[\![f(\varphi_1 \wedge \psi_1^*, \ldots \varphi_k \wedge \psi_k^*)]\!]^{\#} \implies \psi_p$$

therefore

$$[\![e]\!]^{\mathcal{P}\cup\mathcal{P}_r}x \implies \psi_p$$

□

Theorem 25. *Let* $\mathcal{P}_r = \mathsf{OptimizeAndBackPropagate}(p^*, x, y, \mathcal{P}, \mathcal{U})$. *If*

$$\mathcal{L}([\![p^*]\!]^{\mathcal{P}}x, y) < \mathcal{L}([\![p^*]\!]x, y) \text{ or } \mathcal{R}(x, [\![p^*]\!]^{\mathcal{P}}x) < \mathcal{R}(x, [\![p^*]\!]^{(\mathcal{P}\cup\mathcal{P}_r)}x)$$

then

$$\mathcal{L}([\![p^*]\!]^{\mathcal{P}}x, y) < \mathcal{L}([\![p^*]\!]^{(\mathcal{P}\cup\mathcal{P}_r)}x, y) \text{ or } \mathcal{R}(x, [\![p^*]\!]^{\mathcal{P}}x) < \mathcal{R}(x, [\![p^*]\!]^{(\mathcal{P}\cup\mathcal{P}_r)}x)$$

Proof. Let $\varphi = [\![e]\!]^{\mathcal{P}}x$ and ψ^* be the abstract value from which predicates are extracted (line 10, Figure 6-5). If ψ^* was assigned by the if condition (line 9), then

$$\mathcal{L}([\![p^*]\!]^{\mathcal{P}}x, y) < \mathcal{L}(\varphi \wedge \psi^*, y) \text{ or } \mathcal{R}(x, [\![p^*]\!]^{\mathcal{P}}x) < \mathcal{R}(x, \varphi \wedge \psi^*)$$

However if ψ^* was not assigned on line 9, then $\psi^* = (s_0 = p[x])$, and the following is

true:

$$\mathcal{L}([\![p^*]\!]^{\mathcal{P}}x, y) < \mathcal{L}([\![p^*]\!]x, y) = \mathcal{L}(\varphi \wedge \psi^*, y) \text{ or } \mathcal{R}(x, [\![p^*]\!]^{\mathcal{P}}x) < \mathcal{R}(x, [\![p^*]\!]x) = \mathcal{R}(x, \varphi \wedge \psi^*)$$

From Theorem 24,

$$[\![p^*]\!]^{\mathcal{P}\cup\mathcal{P}_r}x \implies \varphi \wedge \psi^*$$

This implies $\mathcal{L}(\varphi \wedge \psi^*, y) \leq \mathcal{L}([\![p^*]\!]^{(\mathcal{P}\cup\mathcal{P}_r)}x, y)$ or $\mathcal{R}(x, \varphi \wedge \psi^*) \leq \mathcal{R}(x, [\![p^*]\!]^{(\mathcal{P}\cup\mathcal{P}_r)}x)$. Therefore $\mathcal{L}([\![p^*]\!]^{\mathcal{P}}x, y) < \mathcal{L}([\![p^*]\!]^{(\mathcal{P}\cup\mathcal{P}_r)}x, y)$ or $\mathcal{R}(x, [\![p^*]\!]^{\mathcal{P}}x) < \mathcal{R}(x, [\![p^*]\!]^{(\mathcal{P}\cup\mathcal{P}_r)}x)$. □

Theorem 26. (Completeness) *Given a dataset $\mathcal{D}$, a DSL G, tolerance $\epsilon \geq 0$, universe of predicates $\mathcal{U}$, initial predicates $\mathcal{P}$, objective function U, loss function $\mathcal{L}$, and the complexity measure C, the Algorithm 6-3 will eventually return some program p_r.*

Proof. Let $\delta_u = \min_{p\in G} U(\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}]), C(p))$, i.e., δ_u is the concrete objective function value of the program which minimizes the objective function.

Let A_i be the FTA constructed in the i^{th} iteration of Algorithm 6-3. Let $\mathcal{P}_i$ be the set of predicates, p_i be the program returned by MinCost in the i^{th} iteration. Let x_i, y_i be the example returned by PickDimension in the i^{th} iteration. Let u_i be the abstract objective function value of p_i with predicates $\mathcal{P}_i$. Note that, in each iteration $u_i \leq \delta_u$ (the abstract loss function is less than the minimum concrete loss function). This implies, there exists δ_I and δ_o, such that,

$$\mathcal{L}([\![p_i]\!]^{\mathcal{P}_i}\boldsymbol{x}, \boldsymbol{x}) \leq \delta_o \text{ and } \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}]) \leq \delta_r$$

Using Theorem 25, if $\mathsf{Distance}(p_i, \boldsymbol{x}_i^*, \mathcal{D}, \mathcal{P}_i) > \epsilon$, then for all $k > i$:

$$\min(\mathcal{L}_o([p_i]\boldsymbol{x}_i, \boldsymbol{y}_c), \mathcal{L}_o([\![p_i]\!]^{\mathcal{P}_i}\boldsymbol{x}_i, \boldsymbol{y}_c) + \delta) \leq \mathcal{L}_o([\![p_i]\!]^{\mathcal{P}_k}\boldsymbol{x}_i, \boldsymbol{y}_c) \leq \mathcal{L}_o([p_i]\boldsymbol{x}_i, \boldsymbol{y}_c)$$

$$\min(\mathcal{R}(\boldsymbol{x}, p_i[\boldsymbol{x}]), \mathcal{R}(\boldsymbol{x}, [\![p_i]\!]^{\mathcal{P}_i}\boldsymbol{x}) + \delta) \leq \mathcal{R}(\boldsymbol{x}, [\![p_i]\!]^{\mathcal{P}_k}\boldsymbol{x}) \leq \mathcal{R}(\boldsymbol{x}, p_i[\boldsymbol{x}])$$

In each iteration, the algorithm picks a program $p \in G$ and increases the abstract loss value or the abstract regularizer value.

Also note that,

$$\mathcal{L}([\![p_i]\!]^{\mathcal{P}_k}\boldsymbol{x}, \boldsymbol{y}) \leq \delta_o + \delta$$

$$\mathcal{R}(\boldsymbol{x}, [\![p_i]\!]^{\mathcal{P}_k}\boldsymbol{x}) \leq \delta_r + \delta$$

because if $\mathcal{L}([\![p_i]\!]^{\mathcal{P}_k}\boldsymbol{x}, \boldsymbol{y}) > \delta_o$ or $\mathcal{R}(\boldsymbol{x}, [\![p_i]\!]^{\mathcal{P}_k}\boldsymbol{x}) > \delta_r$, then the abstract objective function value of p_i is greater than δ_u. In this case, MinCost will never return p_i. Therefore, in worst case MinCost will return a program p at most $\frac{\delta_o}{\delta} + 1 + \frac{\delta_r}{\delta} + 1$ times. The space of programs is finite (due to the restriction of the size of the AFTA). The algorithm will terminate in finite number of iterations. □

Theorem 27. (Correctness) *Given a dataset $\mathcal{D}$, a DSL G, tolerance $\epsilon \geq 0$, universe of predicates $\mathcal{U}$, initial predicates $\mathcal{P}$, objective function U, loss function L, and the complexity measure C, the Algorithm 6-3 will return a program p_r which satisfies the ϵ-correctness condition (Definition 18).*

Proof. From Theorem 26, Algorithm 6-3 will eventually terminate and return a program p_r. From Theorem 23, the returned program p_r will satisfy the *ϵ-correctness* condition. □

6.7 Implementation

I have implemented my synthesis algorithm in a tool called *Rose*. *Rose* is written in Java. The implementation is modular and allows a user to plug-in different DSLs, abstract semantics, loss functions, objective functions, and complexity measures. To support the experiments presented in Section 6.8, we instantiate the *Rose* implementation with the string-processing domain-specific language from [42].

Domain Specific Language and Abstractions: I use the string processing domain specific language from [42, 22] (Figure 6-7), which supports extracting substrings (using the SubStr function) of the input string x and concatenation of substrings (using the Concat function). The function SubStr function extracts a substring using a start and an end position. A position can either be a constant index (ConstPos) or the start or end of the k^{th} occurrence of the match token τ in the input string (Pos).

$$
\begin{array}{rcl}
\text{String expr } e & := & \mathsf{Str}(f) \mid \mathsf{Concat}(f, e); \\
\text{Substring expr } f & := & \mathsf{ConstStr}(s) \mid \mathsf{SubStr}(x, p_1, p_2); \\
\text{Position } p & := & \mathsf{Pos}(x, \tau, k, d) \mid \mathsf{ConstPos}(k); \\
\text{Direction } d & := & \mathsf{Start} \mid \mathsf{End};
\end{array}
$$

Figure 6-7: DSL for string transformations where τ represents a token, k is an integer, and s is a string constant.

Universe of Predicates: I construct a universe of predicates using predicates of the form $\mathsf{len}(s) = i$, where s is a symbol of a type of string and i presents an integer. I also include predicates of the form $s[i] = c$ indicating the i^{th} character of string s is c. Besides these predicates, I also include predicates of the form $s = c$, where c is a value which a symbol s can take. I also include both true and false. In summary, the universe of predicates, I am using, is:

$$
\begin{aligned}
\mathcal{U} = \Big\{\mathsf{len}(s) = i \mid i \in \mathbb{N}\Big\} \cup \Big\{s[i] = c \mid i \in \mathbb{N}, c \in \mathsf{Char}\Big\} \cup \\
\Big\{s = c \mid c \in \mathsf{Type}(s)\Big\} \cup \Big\{\mathsf{true}, \mathsf{false}\Big\}
\end{aligned}
$$

Abstract Semantics: I define a generic transformer for conjunctions of predicates as follows:

$$
f\left((\bigwedge_{i_1} p_{i_1}), \ldots, (\bigwedge_{i_k} p_{i_k})\right) := \prod_{i_1} \ldots \prod_{i_k} f(p_{i_1}, \ldots p_{i_k})
$$

This allows us to just define an abstract semantics for every possible combination of atomic predicates, instead of abstract semantics for all possible abstract values. Figure 6-8 presents the abstract semantics for functions in string processing DSL for all possible combinations of atomic predicates.

Initial Abstraction: The initial abstraction set $\mathcal{P}$ includes predicates of form $\mathsf{len}(s) = i$, where s is a symbol of type string and i is an integer. It also includes true and false.

Abstractions and Loss Functions: I present the abstract version of the $0/\infty$ loss function and $0/1$ loss function below:

$\mathcal{L}_{0/\infty}(\varphi, y) = 0$ if $y \in \gamma(\varphi), \infty$ otherwise and $\mathcal{L}_{0/1}(\varphi, y) = 0$ if $y \in \gamma(\varphi), 1$ otherwise

$$
\begin{array}{rcl}
[\![f(s_1 = c_1, \ldots, s_k = c_k)]\!]^{\#} & := & (s = [\![f(c_1, \ldots c_k)]\!]) \\
[\![\mathsf{Concat}(\mathsf{len}(f) = i_1, \mathsf{len}(e) = i_2)]\!]^{\#} & := & (\mathsf{len}(e) = (i_1 + i_2)) \\
[\![\mathsf{Concat}(\mathsf{len}(f) = i_1, e[i_2] = c)]\!]^{\#} & := & (e[i_1 + i_2] = c) \\
[\![\mathsf{Concat}(\mathsf{len}(f) = i, e = c)]\!]^{\#} & := & (\mathsf{len}(e) = (i + \mathsf{len}(c)) \\
 & & \wedge \bigwedge_{j=1}^{\mathsf{len}(c)} e[i + j - 1] = c[j - 1] \\
[\![\mathsf{Concat}(f[i] = c, p)]\!]^{\#} & := & (e[i] = c) \\
[\![\mathsf{Concat}(f = c, \mathsf{len}(e) = i)]\!]^{\#} & := & (\mathsf{len}(e) = (\mathsf{len}(c) + i)) \\
 & & \wedge \bigwedge_{j=1}^{\mathsf{len}(c)} e[j - 1] = c[j - 1] \\
[\![\mathsf{Concat}(f = c_1, e[i] = c_2)]\!]^{\#} & := & (e[\mathsf{len}(c_1) + i] = c_2) \\
 & & \wedge \bigwedge_{j=1}^{\mathsf{len}(c_1)} e[j - 1] = c_1[j - 1] \\
[\![\mathsf{Str}(p)]\!]^{\#} & := & p
\end{array}
$$

Figure 6-8: Abstract semantics for string transformation DSL.

If $\varphi \neq \mathsf{false}$ ($L_{DL}(\mathsf{false}, y) = \infty$), the abstract version of the Damerau-Levenshtein is $\mathcal{L}_{DL}(\varphi, y) = d_{c,y}(|c|, |y|)$, where $c = \mathsf{ToStr}(\varphi, y)$ and d is defined in Figure 6-9. Let $\mathcal{P} = \mathsf{ExtractPredicates}(\varphi)$. The procedure ToStr returns an array c, such that, if $\mathsf{len}(s) = i \in \mathcal{P}$ then $|c| = i$, otherwise it is the maximum of the length of string y or i such that $s[i] = c' \in \mathcal{P}$. For all $s[i] = c_i \in \mathcal{P}$, $c[i] = c_i$, otherwise it is null.
The abstract version of the 1-Delete loss function:

$$
L_{1D}(\varphi, y) = \begin{cases} 0 & y \in \gamma(\varphi) \\ 1 & a \cdot b = y \text{ and } (\exists c. a \cdot c \cdot b \in \gamma(\varphi) \text{ and } |c| = 1) \\ \infty & \text{otherwise} \end{cases}
$$

The abstract version of the n-Substitution loss function is:

$$
L_{nS}(\varphi, y) = \begin{cases} 0 & \varphi = \mathsf{true} \\ \infty & \varphi = \mathsf{false} \\ L_{nS}(c, y) & \varphi = (s = c) \\ \infty & c_\varphi = \mathsf{ToStr}(\varphi, y) \text{ and } |c_\varphi| \neq |y| \\ \sum_{j=1}^{|y|} \mathbb{1}(c[j] \neq \mathsf{null} \text{ and } c[j] \neq y[i_j]) & c = \mathsf{ToStr}(\varphi, y) \text{ and } |c| = |y| \end{cases}
$$

$$d_{c,y}(i,j) = \min \begin{cases} j & i = 0 \\ i & j = 0 \\ d_{c,y}(i-1,j-1) & i,j > 0 \\ & \text{ and } (c[i-1] = \mathsf{null} \text{ or } c[i-1] = y[j-1]) \\ 1 + d_{c,y}(i-1,j-1) & i,j > 0 \\ & \text{ and } (c[i-1] \neq \mathsf{null} \text{ and } c[i-1] \neq y[j-1]) \\ 1 + d_{c,y}(i-1,j) & i > 0 \\ 1 + d_{c,y}(i,j-1) & j > 0 \\ d_{c,y}(i,j-1) & i = |y| \\ & \text{ and } \varphi \text{ may contain strings of multiple lengths.} \\ 1 + d_{c,y}(i-2,j-2) & i,j > 1 \\ & \text{ and } (c[i-1] = \mathsf{null} \text{ or } c[i-1] = y[i-2]) \\ & \text{ and } (c[i-2] = \mathsf{null} \text{ or } c[i-2] = y[i-1]) \end{cases}$$

Figure 6-9: Abstract semantics for the Damerau-Levenshtein loss function.

6.8 Experimental Results

I use the SyGuS 2018 benchmark suite [3] to evaluate *Rose* against my noisy program synthesis technique presented in Chapter 4.

I run all experiments on a 3.00 GHz Intel(R) Xeon(R) CPU E5-2690 v2 machine with 512GB memory running Linux 4.15.0. I set a timeout limit of 10 minutes for each synthesis task. I compare *Rose* with my tree automaton based noisy program synthesis system presented in Chapter 3, running with a *bounded scope height threshold of* 4 for all experiments.

Noisy Data Sets

Figure 6-10 presents results for all SyGus 2018 benchmark problems which contain less than ten input/output examples. I omit benchmarks for which both *Rose* and tree automata based algorithm timeout (so the rows would contain all -). The first column (Benchmark) presents the name of the SyGus 2018 benchmark. The second column (Number of Input/Output Examples) presents the number of input/output examples in the benchmark problem. The remaining columns present running times, in milliseconds, for *Rose* and tree automata based algorithm running with different

Benchmark	no. of examples	Rose			CFTA
		$\mathcal{L}_{1D}$	$\mathcal{L}_{DL}$	$\mathcal{L}_{0/1}$	
bikes	6	160	175	164	19554
bikes-short	6	160	184	192	21210
dr-name	4	1307	-	1518	-
dr-name-short	4	1560	-	1328	-
firstname	4	256	3993	326	4258
firstname-short	4	334	3917	322	4220
initials	4	19120	19442	316201	36188
initials-short	4	18200	18189	325406	30920
lastname	4	198	195	190	175762
lastname-short	4	194	202	195	178825
name-combine-3	6	4871	4678	5205	547447
name-combine-3-short	6	5622	5515	5671	544044
phone	6	137	151	135	943
phone-short	6	138	135	138	963
phone-1	6	138	135	137	933
phone-1-short	6	149	138	138	942
phone-2	6	140	138	139	953
phone-2-short	6	141	106	136	943
phone-5	7	121	109	113	122
phone-5-short	7	114	89	117	127
phone-6	7	1165	1051	1130	3230
phone-6-short	7	1228	1228	1013	3327
phone-7	7	302	279	278	2793
phone-7-short	7	292	258	298	2762
phone-8	7	296	256	308	3464
phone-8-short	7	275	254	292	3223

Figure 6-10: Runtime performance of *Rose* and CFTA on benchmarks with deletion based noise.

noise sources and loss functions. A - indicates that the corresponding run timed out without synthesizing a program.

The noise source cyclically deletes a single character from outputs in the dataset, starting with the first character, then wrapping around when reaching the last position in the output. The noise source corrupts the first 2 input/output examples in the set of input/output examples. For *Rose*, the figure presents results for each of the $\mathcal{L}_{0/1}, \mathcal{L}_{DL}$, and $\mathcal{L}_{1D}$ loss functions. For my tree automaton based system, I report one running time for each benchmark problem — for this system, the running time is the same for all noise source/loss function combinations.

Benchmark	No of Examples	Rose		CFTA
		$\mathcal{L}_{nS}$	$\mathcal{L}_{DL}$	
phone-long	100	0.71	1.12	32.79
phone-long-repeat	400	2.77	6.20	43.24
phone-1-long	100	0.73	1.15	16.58
phone-1-long-repeat	400	2.51	6.73	17.5
phone-2-long	100	0.63	1.19	16
phone-2-long-repeat	400	2.28	6.84	17.55
phone-5-long	100	0.58	0.79	4.2
phone-5-long-repeat	400	1.64	2.82	14.09
phone-6-long	100	1.14	1.83	103.3
phone-6-long-repeat	400	4.68	9.83	143.9
phone-7-long	100	1.34	1.98	108.1
phone-7-long-repeat	400	4.23	9.90	115.42
phone-8-long	100	1.28	1.89	114.54
phone-8-long-repeat	400	5	10.15	128.3

Figure 6-11: *Rose* and CFTA's performance on dataset corrupted by substitution based noise.

For the benchmarks on which both terminate, *Rose* runs up to 920 times faster than my tree automata based system, with an average speedup of 99 times over tree automata based system. These results highlight the substantial performance benefits that *Rose* delivers.

Every synthesized program is guaranteed to minimize the objective function over the given input/output examples. For a given noise source/loss function combination, my prior system and *Rose* synthesize the same program (unless one or both of the systems times out). These results highlight the ability of *Rose* to synthesize correct programs even in the face of significant noise.

Figure 6-11 presents results for the SyGus 2018 phone-*-long and phone-*-long-repeat benchmarks running with a noise source that cyclically and probabilistically replaces a single digit in each output string with the next digit (wrapping back to 0 if the current digit is 9). I omit benchmarks on which both the techniques timed out. The noise source iterates through each output string in the dataset in turn, probabilistically replacing the next character position in each output string with another character, wrapping around to the first character position when it reaches the

last character position in the output string. The noise source corrupts 95% of the input-output examples in each dataset.

I report results for two loss functions, $\mathcal{L}_{nS}$ (n-substitution) and $\mathcal{L}_{DL}$ (Damerau-Levenshtien). The objective function is the lexicographic objective function. The complexity measure is program size. There is a row in the figure for each phone-*-long-repeat and phone-*-long benchmark; each entry presents the running time (in seconds) for the corresponding synthesis algorithm running on the corresponding benchmark problem.

For the benchmarks on which both terminate, *Rose* runs up to 89 times faster than my prior system, with a median speedup of 27 times over tree automata based system. Once again, these results highlight the substantial performance benefits that *Rose* delivers.

Every synthesized program is guaranteed to minimize the objective function over the given input/output examples. All synthesized programs have zero loss over the original (unseen during synthesis) noise-free input/output examples (i.e., all synthesized programs generate the correct output for each given input). Once again, these results highlight the ability of *Rose* to synthesize correct programs even in the face of significant noise.

Noise-Free Data Sets I have also evaluated the performance of *Rose* and my prior system by applying it to all problems in the SyGuS 2018 benchmark suite [3]. I also compare the performance of *Rose* against with Blaze [42]. Blaze is a programming-by-example synthesis algorithm which uses abstraction refinement to synthesize programs over noise-free data.

For each problem I synthesize the optimal program over clean (noise-free) datasets. I present these result for all SyGuS benchmarks in Figures 6-12 and 6-13.

For *Rose* , the figure presents results for 4 loss functions, zero-one loss function $\mathcal{L}_{0/1}$, Damerau-Levenshtein loss function $\mathcal{L}_{DL}$, 1-Delete loss function $\mathcal{L}_{1D}$, and the n-Substitution loss function $\mathcal{L}_{nS}$.

For the benchmarks on which both *Rose* and CFTA terminate, *Rose* runs up to 1891 times faster than my CFTA based system, with the average speedup of 179 times

Benchmark	No of Examples	Rose				CFTA	Blaze
		$\mathcal{L}_{nS}$	$\mathcal{L}_{1D}$	$\mathcal{L}_{DL}$	$L_{0/1}$	Threshold 4	
bikes	6	71	59	66	58	19554	59
bikes-long	24	293	315	491	253	58187	71
bikes-long-repeat	58	471	585	882	460	127214	67
bikes_small	6	57	59	75	57	21210	58
dr-name	4	891	877	1105	877	-	137
dr-name-long	50	8653	25349	39389	21044	-	267
dr-name-long-repeat	150	23199	82373	214435	61192	-	270
dr-name_small	4	851	791	1057	853	-	140
firstname	4	167	165	167	161	4258	79
firstname-long	54	585	693	1160	6403	37946	290
firstname-long-repeat	204	1886	2323	4635	2012	148101	319
firstname_small	4	165	165	191	163	4220	78
initials	4	20031	19120	19442	316201	36188	118
initials-long	54	102900	108414	133413	98685	378070	338
initials-long-repeat	204	402072	571876	-	428459	-	365
initials_small	4	17877	18221	18189	325406	30920	138
lastname	4	116	110	126	115	175762	90
lastname-long	54	315	304	526	299	565654	320
lastname-long-repeat	204	921	916	2342	894	-	303
lastname_small	4	116	115	128	118	178825	90
name-combine	6	-	-	-	-	-	130
name-combine-long	50	-	-	-	-	-	289
name-combine-long-repeat	204	-	-	-	-	-	398
name-combine_short	6	-	-	-	-	-	127
name-combine-2	4	-	-	-	-	-	228
name-combine-2-long	54	-	-	-	-	-	405
name-combine-2-long-repeat	204	-	-	-	-	-	431
name-combine-2_short	4	-	-	-	-	-	226
name-combine-3	6	4857	5237	8316	4776	547447	150
name-combine-3-long	50	6372	8080	20886	6082	-	280
name-combine-3-long-repeat	200	24988	41678	181297	22893	-	289
name-combine-3_short	6	4964	5041	8599	5049	544044	146
name-combine-4	5	-	-	-	-	-	269
name-combine-4-long	50	-	-	-	-	-	289
name-combine-4-long-repeat	200	-	-	-	-	-	398
name-combine-4_short	5	-	-	-	-	-	127
phone	6	99	96	143	100	943	60
phone-long	100	435	431	1260	431	8592	279
phone-long-repeat	400	1714	1772	4919	1772	28836	281
phone_short	6	98	97	149	97	963	62
phone-1	6	98	255	149	252	933	65
phone-1-long	100	439	3127	1110	2609	8173	295
phone-1-long-repeat	400	1737	15012	6855	7552	28547	284
phone-1_short	6	99	254	145	255	942	79
phone-2	6	98	98	145	94	953	64
phone-2-long	100	431	607	1162	435	6849	301
phone-2-long-repeat	400	1719	3060	5055	1952	29766	293
phone-2_short	6	96	97	148	99	943	66
phone-3	6	-	-	-	-	-	229
phone-3-long	100	-	-	-	-	-	430
phone-3-long-repeat	400	-	-	-	-	-	423
phone-3_short	6	-	-	-	-	-	224
phone-4	6	-	-	-	-	-	249
phone-4-long	100	-	-	-	-	-	441
phone-4-long-repeat	400	-	-	-	-	-	443
phone-4_short	6	-	-	-	-	-	247
phone-5	7	76	78	113	78	122	85
phone-5-long	100	367	375	831	368	683	371
phone-5-long-repeat	400	1096	1257	2840	1155	2179	380
phone-5_short	7	78	77	118	76	127	86
phone-6	7	192	194	271	190	3230	91
phone-6-long	100	892	1049	1868	922	27566	386
phone-6-long-repeat	400	2990	4412	9976	3128	100241	377
phone-6_short	7	195	189	292	192	3327	90
phone-7	7	184	221	329	216	2793	91
phone-7-long	100	904	1059	1887	894	27770	381
phone-7-long-repeat	400	2852	4619	7282	2845	89535	377
phone-7_short	7	188	214	269	185	2762	90
phone-8	7	220	189	310	185	3464	92
phone-8-long	100	923	1030	1999	954	22961	356
phone-8-long-repeat	400	2817	4709	10768	3336	88932	388
phone-8_short	7	221	192	312	183	3223	93

Figure 6-12: Runtime performance of *Rose* , my CFTA based system, and Blaze over noise-free dataset.

over my CFTA based system. *Rose* is also able to synthesize the correct program for 8 more benchmarks compared to my tree automaton based system.

For the benchmark on which both *Rose* and Blaze terminate, Blaze runs, on an

Benchmark	No of Examples	Rose				CFTA	Blaze
		$\mathcal{L}_{nS}$	$\mathcal{L}_{1D}$	$\mathcal{L}_{DL}$	$L_{0/1}$	Threshold 4	
phone-9	7	-	-	-	-	-	1338
phone-9-long	100	-	-	-	-	-	1689
phone-9-long-repeat	400	-	-	-	-	-	1783
phone-9_short	7	-	-	-	-	-	1346
phone-10	7	-	-	-	-	-	2246
phone-10-long	100	-	-	-	-	-	2427
phone-10-long-repeat	400	-	-	-	-	-	2358
phone-10_short	7	-	-	-	-	-	2302
reverse-name	6	-	-	-	-	-	128
reverse-name-long	50	-	-	-	-	-	284
reverse-name-long-repeat	200	-	-	-	-	-	274
reverse-name_short	6	-	-	-	-	-	131
univ_1	6	-	-	-	-	-	1568
univ_1-long	20	-	-	-	-	-	2395
univ_1-long-repeat	30	-	-	-	-	-	1285
univ_1_short	6	-	-	-	-	-	1370
univ_2	6	-	-	-	-	-	558316
univ_2-long	20	-	-	-	-	-	-
univ_2-long-repeat	30	-	-	-	-	-	-
univ_2_short	6	-	-	-	-	-	522689
univ_3	6	-	-	-	-	-	4621
univ_3-long	20	-	-	-	-	-	-
univ_3-long-repeat	30	-	-	-	-	-	-
univ_3_short	6	-	-	-	-	-	4639
univ_4	8	-	-	-	-	-	-
univ_4-long	20	-	-	-	-	-	-
univ_4-long-repeat	30	-	-	-	-	-	-
univ_4_short	8	-	-	-	-	-	-
univ_5	8	-	-	-	-	-	-
univ_5-long	20	-	-	-	-	-	-
univ_5-long-repeat	30	-	-	-	-	-	-
univ_5_short	8	-	-	-	-	-	-
univ_6	8	-	-	-	-	-	-
univ_6-long	20	-	-	-	-	-	-
univ_6-long-repeat	30	-	-	-	-	-	-
univ_6_short	8	-	-	-	-	-	-

Figure 6-13: Runtime performance of *Rose* , my CFTA based system, and Blaze over noise-free dataset.

average, 72 times faster compared to *Rose*, with the median speedup of 3 times over *Rose*. Blaze is also able to synthesize 40 more benchmarks compared to *Rose*.

Both *Rose* and Blaze use the refinement steps to identify correct and incorrect programs. For Blaze, indefying incorrect programs is very efficient. If a program does not satisfy even a single input-output example, Blaze can prune that program from the search space. This allows them to aggresively prune the search space of programs. Because our technique works with noisy datasets, it can never tell if a program has minimal objective function value without comparing the program to all programs in our search space (unless the loss happens to be zero). My technique can only prune a program, if and only if, it can prove that the this program's objective function value is greater than the minimal objective function value. The difference in how aggressively these techniques prune the search space explain the difference in performance between these techniques.

6.9 Discussion

I present a new technique to synthesize programs over noisy datasets. This technique uses an abstraction refinement based optimization process to search for a program which *best-fits* a given dataset, based on an objective function, a loss function, and a complexity measure.

I have implemented my synthesis algorithm in the *Rose* noisy program synthesis system. My experimental results show that, on two noisy benchmark program synthesis problem sets drawn from the SyGus 2018 benchmarks, *Rose* delivers significant speedups over my tree automaton based technique.

Chapter 7

Domain Specific Languages With Infinite Sets of Constants

Using a small finite set of constants is a standard practice in enumeration based program synthesis techniques [43, 42, 3]. I present a modification to our abstraction refinement based algorithm which allows it to handle domain specific languages with an infinite sets of constants. This chapter builds up on the notation and concepts introduced in Chapter 6.

I first formalize domain specific languages with an infinite sets of constants. I do this by associating each terminal symbol with a set of values that the terminal symbol can be replaced with in a program. I formalize complexity measures over programs accepted by these DSLs. Given a program, these complexity measures assign a weight to a constant value based on the terminal it replaces. I then use these concepts to modify the algorithm presented in Chapter 6 to work with these concepts.

The core modification is to partition the space of constants into abstract values. These abstract values are then used to build an abstract finite tree automaton. This AFTA takes the constant abstract values and partitions the program space based on these abstract values. The algorithm then proceeds in a manner similar to my original algorithm. The algorithm synthesizes a program which minimizes the abstract objective function. If the difference between the concrete objective function value and the abstract objective function value of this program is not greater than the tolerance

value, the algorithm returns this program. If the difference is greater than tolerance value, the algorithm then finds a counter example and uses the abstraction refinement based optimization technique to further refine the abstract finite tree automaton and the partitions of space of constants.

My algorithm is guaranteed to terminate. For my experiments, this tolerance is set to 0. In this case, when the algorithm terminates, it synthesizes the program which minimizes the objective function.

7.1 Framework

I first modify our framework to work with domain specific languages with an infinite set of constants and and complexity measures defined over such DSLs.

Domain Specific Languages:

Our framework starts with a domain specific language (DSL) $G = (T, N, P, s_0)$, where $T_C \subset T$ is the set of constant terminals in G. Each constant terminal $t \in T_C$ is associated with a set of (possibly infinite) values (denoted by V_t). I use the notation v_t to denote values from the set V_t (value v of type t).

Complexity Measures:

Given a DSL G, for every constant terminal $t \in T_C$, I assume that we are given a weight function w_t. For all $v \notin V_t$. $w_t(v) = \infty$. My framework allows for complexity measures which are computed recursively on a given program's parse tree. I incorporate weight functions within Cost style complexity measures as follow:

$$\begin{aligned} \text{Cost}(x) &= \text{cost}(x) \\ \text{Cost}(v_t) &= \text{cost}(t) \times w_t(v) \\ \text{Cost}(f(e_1, e_2, \ldots e_k)) &= \text{cost}(f) + \sum_{i=1}^{k} \text{Cost}(e_i) \end{aligned}$$

where t and f are terminals and built-in functions in the DSL respectively, and v_t is a value from set V_t.

I assume, given an abstract value φ, we can compute the constant value $v \in \gamma(\varphi)$

which minimizes the weight function w_t, i.e.,

$$v^* = \arg\min_{v \in \gamma(\varphi)} w_t(v)$$

Assumptions: Given a noisy dataset $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$, loss function $\mathcal{L}$, regularizer $\mathcal{R}$, and complexity measure C, we assume that the objective function U satisfies the following constraint:

$$\forall \delta \geq 0,\ \exists \delta_o \geq 0, \delta_r \geq 0, \delta_c \geq 0. U(\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}]), C(p)) \leq \delta$$

$$\implies \mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}_c) \leq \delta_o, \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}]) \leq \delta_r, \text{ and } C(p) \leq \delta_c$$

i.e., if the objective function value is finite, then the loss, the regularizer weight, and the complexity are all finite.

I assume if a predicate $\psi \in \mathcal{U}$, then $\neg\psi \in \mathcal{U}$, i.e, if a predicate is present in the universe of predicates, then its negation also exists within the universe of predicates.

7.2 Synthesis Algorithm

Figure 7-1 presents my modified abstraction refinement based synthesis algorithm. The original synthesis algorithm uses predicates $\mathcal{P}$ to construct an abstract value containing concrete values for each constant. This algorithm instead uses predicates $\mathcal{P}$ to partition the space of all constants. The function GenerateConstantMap, given predicates $\mathcal{P}$, constructs a map C, which maps constant terminals $t \in T_C$ in the DSL G to a set of abstract values (line 3). All of these partitions are valid abstract values for constructing the Abstract Finite Tree Automaton (line 4). Figure 7-2 presents the modified rules to construct an AFTA. Note the Const rule change. Given a constant terminal $t \in T_C$ and the map C, for each abstract value in set $C[t]$, I construct a state q_t^{φ} in the AFTA. The AFTA generated by these rules will accept all programs p accepted by the DSL G.

The algorithm then (similar to the original algorithm) selects a MinCost program (using an updated LeastComplex function). If the distance between the abstract objective function value and the concrete objective function value for the min cost

1: **procedure** SYNTHESIZE($\mathcal{D}, G, \epsilon, \mathcal{P}, \mathcal{U}$)
input: noisy dataset $\mathcal{D} = (\boldsymbol{x}, \boldsymbol{y})$, DSL G, and tolerance ϵ.
input: initial predicates $\mathcal{P}$, and universe of predicates $\mathcal{U}$.
output: A program p, such that, p satisfies the ϵ-*correctness* condition.
2: **while** true **do**
3: $C :=$ GenerateConstantMap$(G, \mathcal{P})$;
4: $\mathcal{A} :=$ ConstructAFTA$(\boldsymbol{x}, G, \mathcal{P}, C)$;
5: $p :=$ MinCost$(\mathcal{A}, \mathcal{D})$;
6: **if** Distance$(p, \mathcal{D}, \mathcal{P}) \leq \epsilon$ **then return** p;
7: $x, y :=$ PickDimension$(p, \mathcal{D}, \mathcal{P})$;
8: $\mathcal{P} := \mathcal{P} \bigcup$ OptimizeAndBackPropogate$(p, x, y, \mathcal{P}, \mathcal{U})$;

Figure 7-1: Algorithm for noisy program synthesis using abstraction refinement based optimization with abstractions for constants.

$$\frac{\boldsymbol{\varphi} = \alpha^{\mathcal{P}}\Big(\langle x = x_1, \ldots x = x_n\rangle\Big)}{q_x^{\boldsymbol{\varphi}} \in Q}\ (\textsc{Var}) \quad \frac{q_{s_0}^{\boldsymbol{\varphi}} \in Q}{q_{s_0}^{\boldsymbol{\varphi}} \in Q_f}\ (\textsc{Final})$$

$$\frac{t \in T_C, \boldsymbol{\varphi} = \langle \varphi, \ldots \varphi\rangle, |\boldsymbol{\varphi}| = n, \varphi \in C[t]}{q_t^{\boldsymbol{\varphi}} \in Q}\ (\textsc{Const})$$

$$\frac{\begin{array}{c} s \to f(s_1, \ldots, s_k) \in P, q_{s_1}^{\boldsymbol{\varphi}_1}, \ldots, q_{s_k}^{\boldsymbol{\varphi}_k} \in Q, \\ \varphi_j = \alpha^{\mathcal{P}}\Big([\![f(\boldsymbol{\varphi}_{1j}, \ldots, \boldsymbol{\varphi}_{kj})]\!]^{\#}\Big), \boldsymbol{\varphi} = \langle \varphi_1, \ldots \varphi_n\rangle \end{array}}{q_s^{\boldsymbol{\varphi}} \in Q, f(q_{s_1}^{\boldsymbol{\varphi}_1}, \ldots, q_{s_k}^{\boldsymbol{\varphi}_k}) \to q_s^{\boldsymbol{\varphi}} \in \Delta}\ (\textsc{Prod})$$

Figure 7-2: Rules for constructing FTA $\mathcal{A} = (Q, F, Q_f, \Delta)$ with Abstract Values, for inputs $\boldsymbol{x} = \langle x_1, \ldots x_n\rangle$ and C, a map from terminals to a set of abstract values.

program is less than ϵ, we are done. If not, the algorithm then (similar to the original algorithm) uses abstraction refinement to increase the set of predicates. I use this expanded set of predicates to refine the abstract values for the AFTA similar to my original algorithm presented in Chapter 6. I also use these predicates to further partition the space of all constants.

GenerateConstantMap: For each t, this function uses AddConstantAbstractions (Figure 7-3) to partition the space of constants. Given predicates $\mathcal{P}$, the method uses all predicates in $\mathcal{P}$ to construct these partitions. For each partition φ (specified via the abstract value φ) and each predicate $\psi \in \mathcal{P}$, either ψ is true for all values in $\gamma(\varphi)$ or false for all values in $\gamma(\varphi)$. The procedure recursively computes the following set

1: **procedure** AddConstantAbstractions($\varphi, \mathcal{P}$)
2: **if** $\mathcal{P} = \emptyset$ **then**
3: **return** $\{\varphi\}$;
4: $\psi \in \mathcal{P}$;
5: $S_p :=$ AddConstantAbstractions$(\varphi \wedge \psi, \mathcal{P} - \{\psi\})$;
6: $S_n :=$ AddConstantAbstractions$(\varphi \wedge \neg\psi, \mathcal{P} - \{\psi\})$;
7: **return** $S_p \cup S_n$;
8: **procedure** GenerateConstantMap($G, \mathcal{P}$)
input: DSL G and predicates $\mathcal{P}$.
output: Map C from constants in G to sets of abstract values.
9: **for** $t \in T_C$ **do**
10: $C[t] :=$ AddConstantAbstractions$(\mathsf{true}, \mathcal{P})$;
11: $C[t] := C[t] \cup \mathsf{false}$;
12: **return** C;

Figure 7-3: Algorithm for generating map from constants to set of abstract values.

of partitions:

$$S = \left\{ \varphi_1 \wedge \ldots \varphi_k \mid \varphi_1 \in \{\psi_1, \neg\psi_1\}, \ldots \varphi_k \in \{\psi_k, \neg\psi_k\}, \mathcal{P} = \{\psi_1, \ldots \psi_k\} \right\}$$

Least Complex Program Algorithm: The least complex program algorithm is similar to the algorithm presented in Chapter 3. The modified algorithm, given state $q_t^{\langle\varphi,\ldots\varphi\rangle}$ generates an expression using the abstract value v^* instead of generating expression terminal t (line 4-6). Here v^* is the value that minimizes the weight function w_t, given abstract value φ.

On Proof of Correctness:

The proof of correctness for this algorithm is identical to the proof presented for my original abstraction refinement based program synthesis algorithm (Chapter 6). The changes in the DSL only effect the proofs for Theorem 21 and Theorem 26. I present the modified proof below:

Theorem 28. (Constant Maps) *Given a set of predicates $\mathcal{P}$, DSL G, and constant map $C :=$ GenerateConstantMap$(\mathcal{P}, G)$, for all terminals t, and all possible values v_t, there exists an abstract value $\varphi \in C[t]$, such that, $v_t \in \gamma(\varphi)$.*

Proof. Given a terminal t and value v_t. Let $\varphi \in \alpha^{\mathcal{P}}(t = v_t)$. φ. Note that, $v_t \in \gamma(\varphi)$.

1: **procedure** LEASTCOMPLEX($q_s^{\boldsymbol{\varphi}}, \mathcal{A}, G$)
input: State $q_s^{\boldsymbol{\varphi}}$, AFTA $\mathcal{A} = (Q, Q_f, \Delta)$, and DSL G.
input: Recursively defined complexity measure cost.
output: Least complex expression e^* and it's complexity c^*.
2: **if** $s = x$ **then**
3: **return** $x, \mathsf{cost}(x)$
4: **if** $s \in T_C$ **then**
5: $v^* := \arg\min_{v \in \boldsymbol{\varphi}_0} w_s(v)$
6: **return** $v^*, \mathsf{cost}(s) \times w_s(v^*)$
7: $c^* := \infty$
8: **for** $f(q_{s_1}^{\boldsymbol{\varphi}_1}, \ldots, q_{s_k}^{\boldsymbol{\varphi}_k}) \to q_s^{\boldsymbol{\varphi}} \in \Delta$ **do**
9: **for** $i := 1 \ldots k$ **do**
10: $e_i, c_i := \mathsf{LeastComplex}(q_{s_i}^{\boldsymbol{\varphi}_i}, \mathcal{A}, G, \mathcal{M})$
11: $e := f(e_1, \ldots e_k)$
12: $c := \mathsf{cost}(f) + \sum_{i=1}^{n} c_i$
13: **if** $c < c^*$ **then**
14: $e^* := e$
15: $c^* := c$
16: **return** e^*, c^*

Figure 7-4: Algorithm for synthesizing a least complex program for automaton $\mathcal{A}$, DSL G, and state $q_s^{\boldsymbol{\varphi}}$.

By negation, φ must contain all predicates from $\mathcal{P}$.

Since the procedure GenerateConstantMap generates all possible abstract values which contain all predicates in set $\mathcal{P}$, therefore, $\varphi \in C[t]$.

Hence, $v_t \in \gamma(\alpha^{\mathcal{P}}(t = v_t))$ and $\alpha^{\mathcal{P}}(t = v_t) \in C[t]$. □

Theorem 29. (Structure of the Tree Automaton) *Given a set of predicates $\mathcal{P}$, input vector $\boldsymbol{x} = \langle x_1, \ldots x_n \rangle$, and DSL G, let $\mathcal{A} = (Q, Q_f, \Delta)$ be the AFTA returned by the function* ConstructAFTA$(\boldsymbol{x}, G, \mathcal{P})$. *Then for all symbols s in G, for all expressions e starting from symbol s (and height less than bound b), there exists a state $q_s^{\boldsymbol{\varphi}} \in Q$, such that, e is accepted by the automaton $(Q, \{q_s^{\boldsymbol{\varphi}}\}, \Delta)$, where $\boldsymbol{\varphi} = \langle [\![e]\!]^{\mathcal{P}} x_1, \ldots [\![e]\!]^{\mathcal{P}} x_n \rangle$.*

Proof. I prove this theorem by using induction over the height of the expression e.

Base Case: Height of expression e is 1. This implies the symbol is either x or a constant. According to Var rule (Figure 7-2), there exists a state $q_x^{\boldsymbol{\varphi}} \in Q$ (for terminal x), where $\boldsymbol{\varphi} = \langle [\![x]\!]^{\mathcal{P}} x_1, \ldots [\![x]\!]^{\mathcal{P}} x_n \rangle$ and x is accepted by automaton $(Q, \{q_x^{\boldsymbol{\varphi}}\}, \Delta)$.

According to Const rule (Figure 7-2), there exists some state $q_t^{\boldsymbol{\varphi}} \in Q$ (for constant terminal t), where $\boldsymbol{\varphi} = \langle \varphi, \ldots \varphi \rangle$, for all $\varphi \in C[t]$. Note that, using Theorem 28, for all possible concrete values v_t for terminal t, there exists an abstract value $\varphi \in C[t]$, such that, $v_t \in \gamma(\varphi)$. Therefore, for all possible expressions e of the form v_t (value v of type v_t, there exists a state $q_t^{\langle \varphi, \ldots \varphi \rangle} \in Q$, where $v_t \in \gamma(\varphi)$ and v_t is accepted by $(Q, \{q_x^{\boldsymbol{\varphi}}\}, \Delta)$.

Induction argument: The induction argument is identical to the proof of DSLs with finite constants (Theorem 21). □

Theorem 30. (Completeness) *Given a dataset $\mathcal{D}$, a DSL G, tolerance $\epsilon \geq 0$, universe of predicates $\mathcal{U}$, initial predicates $\mathcal{P}$, objective function U, loss function $\mathcal{L}$, and the complexity measure C, the Algorithm 6-3 will eventually return some program p_r.*

Proof. Let $\delta_u = \min_{p \in G} U(\mathcal{L}(p[\boldsymbol{x}], \boldsymbol{y}), \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}]), C(p))$, i.e., δ_u is the concrete objective function value of the program which minimizes the objective function.

Let A_i be the FTA constructed in the i^{th} iteration of Algorithm 6-3. Let $\mathcal{P}_i$ be the set of predicates and p_i be the program returned by MinCost in the i^{th} iteration. Let x_i, y_i be the example returned by PickDimension in the i^{th} iteration. Let u_i be the abstract objective function value of p_i with predicates $\mathcal{P}_i$. Note that, in each iteration $u_i \leq \delta_u$ (the abstract loss function is less than the minimum concrete loss function). This implies, there exists δ_o, δ_r, and δ_c, such that,

$$\mathcal{L}([\![p_i]\!]^{\mathcal{P}_i}\boldsymbol{x}, \boldsymbol{x}) \leq \delta_o, \mathcal{R}(\boldsymbol{x}, p[\boldsymbol{x}]) \leq \delta_r, \text{ and } C(p_i) \leq \delta_c$$

Using Theorem 25, if $\mathsf{Distance}(p_i, \boldsymbol{x}_i^*, \mathcal{D}, \mathcal{P}_i) > \epsilon$, then for all $k > i$:

$$\min(\mathcal{L}_o([p_i]\boldsymbol{x}_i, \boldsymbol{y}_c), \mathcal{L}_o([\![p_i]\!]^{\mathcal{P}_i}\boldsymbol{x}_i, \boldsymbol{y}_c) + \delta) \leq \mathcal{L}_o([\![p_i]\!]^{\mathcal{P}_k}\boldsymbol{x}_i, \boldsymbol{y}_c) \leq \mathcal{L}_o([p_i]\boldsymbol{x}_i, \boldsymbol{y}_c)$$

$$\min(\mathcal{R}(\boldsymbol{x}, p_i[\boldsymbol{x}]), \mathcal{R}(\boldsymbol{x}, [\![p_i]\!]^{\mathcal{P}_i}\boldsymbol{x}) + \delta) \leq \mathcal{R}(\boldsymbol{x}, [\![p_i]\!]^{\mathcal{P}_k}\boldsymbol{x}) \leq \mathcal{R}(\boldsymbol{x}, p_i[\boldsymbol{x}])$$

In each iteration, the algorithm picks a program $p \in G$ and increases the abstract loss value or the abstract regularizer value.

Also note that,

$$\mathcal{L}([\![p_i]\!]^{\mathcal{P}_k}\boldsymbol{x}, \boldsymbol{y}) \leq \delta_o + \delta$$

and

$$\mathcal{R}(\boldsymbol{x}, [\![p_i]\!]^{\mathcal{P}_k}\boldsymbol{x}) \leq \delta_r + \delta$$

because if $\mathcal{L}([\![p_i]\!]^{\mathcal{P}_k}\boldsymbol{x}, \boldsymbol{y}) > \delta_o$ or $\mathcal{R}(\boldsymbol{x}, [\![p_i]\!]^{\mathcal{P}_k}\boldsymbol{x}) > \delta_r$, then the abstract objective function value of p_i is greater than δ_u. In this case, MinCost will never return p_i.

Therefore, in the worst case MinCost will return a program p, $\frac{\delta_o}{\delta} + 1 + \frac{\delta_r}{\delta} + 1$ times.

The space of programs with objective function value less than δ_u is finite (due to the restriction of the size of the AFTA and there are only finite programs with $C(p) \leq \delta_c$). The algorithm will terminate in finite number of iterations. □

7.3 Implementation

I have implemented my modified algorithm within the *Rose* synthesis tool. *Rose* is parameterized over a large class of objective functions, loss functions, and complexity measures. I benchmark *Rose* using the SyGuS 2018 benchmark suite [1]. Each benchmark is associated with a finite number of string constants (between 0 to 2). For my experiments, I allow the synthesis algorithm to use all possible strings, i.e., the synthesis algorithm can synthesize programs with arbitrary strings as constants.

Abstractions: I use the universe of predicates, initial abstractions, and abstract loss function semantics presented in Chapter 6 (Section 6.7). I present an expanded version of abstract semantics for my DSL in Figure 7-5.

7.4 Experimental Results

I use the SyGuS 2018 benchmark suite [1] to benchmark by technique. I use the size complexity measure Size(p) (Subsection 2.2.4) and uniform regularizer $\mathcal{R}_U$ (Definition 4) for these experiments. For these experiments, each constant string is given weight 1, i.e., $w_p(s) = 1$, for all string s (p is a constant terminal in my DSL).

7.4.1 Scalability

I evaluate the scalability of my implementation by applying it to all problems in the SyGuS 2018 benchmark suite [1]. For each problem, I use clean (noise-free) data set

$$
\begin{array}{rcl}
[\![f(s_1 = c_1, \ldots, s_k = c_k)]\!]^{\#} & := & (s = [\![f(c_1, \ldots c_k)]\!]) \\
[\![\mathsf{Concat}(\mathsf{len}(f) = i_1, \mathsf{len}(e) = i_2)]\!]^{\#} & := & (\mathsf{len}(e) = i_1 + i_2) \\
[\![\mathsf{Concat}(\mathsf{len}(f) = i_1, e[i_2] = c)]\!]^{\#} & := & (e[i_1 + i_2] = c) \\
[\![\mathsf{Concat}(\mathsf{len}(f) = i, e = c)]\!]^{\#} & := & (\mathsf{len}(e) = (i + \mathsf{len}(c)) \\
 & & \wedge \bigwedge_{j=1}^{\mathsf{len}(c)} e[i + j - 1] = c[j - 1] \\
[\![\mathsf{Concat}(f[i] = c, p)]\!]^{\#} & := & (e[i] = c) \\
[\![\mathsf{Concat}(f = c, \mathsf{len}(e) = i)]\!]^{\#} & := & (\mathsf{len}(e) = (\mathsf{len}(c) + i)) \\
 & & \wedge \bigwedge_{j=1}^{\mathsf{len}(c)} e[j - 1] = c[j - 1] \\
[\![\mathsf{Concat}(f = c_1, e[i] = c_2)]\!]^{\#} & := & (e[\mathsf{len}(c_1) + i] = c_2) \\
 & & \wedge \bigwedge_{j=1}^{\mathsf{len}(c_1)} e[j - 1] = c_1[j - 1] \\
[\![\mathsf{Str}(p)]\!]^{\#} & := & p \\
[\![\mathsf{ConstStr}(p)]\!]^{\#} & := & p
\end{array}
$$

Figure 7-5: Abstract semantics for string transformation DSL.

for the problem provided with the benchmark suite. While running these benchmarks, I do not provide any constants (given in the SyGuS benchmark) to my algorithm. All experiments are run on an 3.00 GHz Intel(R) Xeon(R) CPU E5-2690 v2 machine with 512GB memory running Linux 4.15.0. With a timeout limit of 10 minutes and bounded scope height threshold of 4, the implementation is able to solve 49 out of the 108 SyGuS 2018 benchmark problems. For the remaining benchmark problems, *Rose* times out.

Figure 7-6 presents my results for SyGuS 2018 benchmarks. There is a row for each benchmark problem. The first column presents the name of the benchmark. The next row presents the number of examples in that benchmark. The next four rows present performance of the technique running on four different loss functions in milliseconds. These loss functions are n-Substitution loss function $\mathcal{L}_{nS}$, 1-Delete loss function $\mathcal{L}_{1D}$, Damerau-Levenstein loss function $\mathcal{L}_{DL}$, and 0/1 loss function $\mathcal{L}_{0/1}$.

7.4.2 Noisy Data Sets, Character Deletions

I next present results for my implementation running on SyGuS benchmarks with number of input-output examples less than 10 with character deletions. I omit benchmarks on which my implementation fails to terminate. I use a noise source that cycli-

Benchmark Name	Number of Examples	$\mathcal{L}_{nS}$	$\mathcal{L}_{1D}$	$\mathcal{L}_{DL}$	$\mathcal{L}_{0/1}$
bikes	6	156	161	234	160
bikes-long	24	288	301	438	301
bikes-long-repeat	58	544	646	1130	542
bikes_small	6	157	159	233	159
dr-name	4	147056	217658	233509	229024
dr-name-long	50	185278	214971	492378	196555
dr-name-long-repeat	150	552474	-	-	596445
dr-name_small	4	146017	225172	230840	216359
firstname	4	209	206	213	212
firstname-long	54	925	1028	1839	1049
firstname-long-repeat	204	3713	3649	10727	3479
firstname_small	4	209	208	215	206
initials	4	-	-	-	276582
initials_small	4	-	-	-	271477
lastname	4	63	62	68	62
lastname-long	54	166	179	416	170
lastname-long-repeat	204	394	430	1532	390
lastname_small	4	62	87	68	63
name-combine-3	6	81069	79390	89956	84482
name-combine-3-long	50	399983	430599	-	397408
name-combine-3_short	6	80402	84738	85040	78127
phone	6	89	88	89	91
phone-long	100	415	498	717	420
phone-long-repeat	400	1239	3021	4952	1144
phone_short	6	86	90	89	135
phone-1	6	119	243	122	241
phone-1-long	100	444	502	713	410
phone-1-long-repeat	400	1220	2539	4961	1139
phone-1_short	6	119	242	122	138
phone-2	6	120	121	122	138
phone-2-long	100	660	863	1235	650
phone-2-long-repeat	400	2130	4123	8601	1903
phone-2_short	6	126	118	120	106
phone-5	7	124	128	128	109
phone-5-long	100	699	721	924	674
phone-5-long-repeat	400	2290	3083	5235	2308
phone-5_short	7	121	121	129	89
phone-6	7	900	893	934	1051
phone-6-long	100	8915	7946	12537	7220
phone-6-long-repeat	400	31729	42420	108900	31079
phone-6_short	7	876	878	945	1228
phone-7	7	1063	990	1122	279
phone-7-long	100	8186	8495	13965	8616
phone-7-long-repeat	400	35519	41088	120519	32427
phone-7_short	7	1057	892	1191	258
phone-8	7	1000	903	961	256
phone-8-long	100	6942	8109	12695	7214
phone-8-long-repeat	400	31711	40965	107993	32111
phone-8_short	7	882	907	978	254

Figure 7-6: Runtime performance of *Rose* over noise-free dataset.

cally deletes a single character from each output in the dataset in turn. I consider three loss functions: the 0/1 loss function, Damerau-Levenstein loss function, and the 1-Delete loss function.

Figure 7-7 summarizes these results. There is a row for each benchmark problem. The first column presents the name of the benchmark. The next row presents the number of examples in that benchmark. The next three rows present the minimum number of correct input-output examples are required for the synthesis technique to synthesize the correct program, given different loss functions. These loss functions are 1-Delete loss function $\mathcal{L}_{1D}$, Damerau-Levenstein loss function $\mathcal{L}_{DL}$, and 0/1 loss function $\mathcal{L}_{0/1}$. Note that these results are consistent with results presented in Chapter 4.

Benchmark	Number of examples Size	Number of Required Correct Examples		
		1-Delete	DL	0/1
bikes	6	0	0	3
dr-name	4	0	0	2
firstname	4	0	0	2
lastname	4	0	1	1
name-combine-3	6	0	0	2
phone	6	0	2	3
phone-1	6	0	3	3
phone-2	6	0	2	3
phone-5	7	0	2	3
phone-6	7	0	1	3
phone-7	7	0	2	3
phone-8	7	0	0	1

Figure 7-7: Minimum number of correct examples required to synthesize a correct program.

7.5 Discussion

I present a new technique to synthesize programs over noisy datasets, given a domain specific languages with a large (possibly infinite) set of constants. I have implemented my synthesis algorithm in the *Rose*. My experimental results show that our system can search for optimal constants to synthesize the correct program, even when these constants are not provided to my algorithm.

Chapter 8

Dealing With Noisy Inputs

I now generalize the noisy program synthesis algorithm to work with noisy datasets that include noisy inputs in addition to noisy outputs. Noisy inputs create a novel problem within the synthesis process. Since the synthesis algorithm does not have access to noise-free inputs, the synthesis algorithm cannot compute the corresponding noise-free outputs. The noise-free outputs are required for computing the loss of a program over the noisy dataset. Due to the potential discrete nature of programs in our program space, using the noisy inputs may not help in computing the output loss (i.e., small changes in the input may lead to large differences in a discrete program's output). I solve this problem by not only synthesizing a *best-fit* program, but also synthesizing the *best-fit* noise-free inputs. I introduce the additional concept of input loss function, which measures the loss between the noisy inputs and the best-fit noise-free inputs. The objective function is modified to incorporate the input loss function, thus allowing me to frame the synthesis problem over datasets with noisy inputs and noisy outputs as an optimization problem.

I then modify the algorithm presented in Chapter 6 to work with this new optimization problem. The algorithm's refinement loop starts by partitioning the space of inputs into input partitions, represented using abstract values. These abstract values are used as abstract input vector to build an abstract finite tree automaton. Note that the abstract input vector may not be of the same length as the noisy input vector. For each accepting state, the algorithm first computes the simplest program

accepted by the state. The output abstract values, attached with each state, are the equal to the abstract output value returned by this program on the input partitions. The algorithm then assigns each noisy input-output pair an input partition-abstract output value pair, such that, this assignment makes it possible to minimize the abstract objective function. The algorithm then iterates through all accepting states and computes the program and noisy input and input partition assignment which minimizes the abstract objective function. Using this assignment and the input loss function, for each noisy input, the algorithm computes the corresponding noise-free input. For a noisy input, the corresponding noise-free input is the input, within the noisy input's assigned input partition, which minimizes the input loss value between the noise-free input and the noisy input.

The algorithm computes the concrete objective function value of the candidate program and candidate noise-free inputs. If the difference between the concrete objective function value and the abstract objective function value of this program is not greater than the tolerance value, the algorithm returns this program. If the difference is greater than the tolerance value, the algorithm then finds a counter example. The counter example, in this case, is a pair of a noise-free input and the corresponding noisy output, such that, the output loss between the noise-free output (candidate program's output on the noise-free input) and the noisy output is greater than 0. The algorithm then uses this counter example and abstract refinement based optimization technique to further refine the abstract finite tree automaton and the partitions of space of inputs.

My algorithm is guaranteed to terminate. For my experiments, this tolerance is set to 0. In this case, when the algorithm terminates, it synthesizes the program which minimizes the objective function.

8.1 Framework

I first modify the concept of a noisy dataset (introduced in Chapter 2) to include the concept of noisy inputs.

Noisy Dataset:

A noisy dataset $\mathcal{D} = (\boldsymbol{x}_c, \boldsymbol{y}_c)$ is composed of a set of noisy inputs $\boldsymbol{x}_c$ and corresponding noisy outputs $\boldsymbol{y}_c$. I assume the dataset $\mathcal{D} = (\boldsymbol{x}_c, \boldsymbol{y}_c)$ of size n is constructed by the following process:

- A hidden program $p_h \in G$ is randomly picked from the set of programs G (defined using a domain specific language).
- n hidden inputs $\boldsymbol{x}$ are sampled from the input source.
- Hidden outputs $\boldsymbol{z} = p_h[\boldsymbol{x}]$ are computed.
- A noise source ρ_{N_i} corrupts the inputs $\boldsymbol{x}$ to noisy inputs $\boldsymbol{x}_c$ and a noise source ρ_{N_o} corrupts outputs $\boldsymbol{z}$ to noisy outputs $\boldsymbol{y}_c$.

The synthesis algorithm is given the noisy inputs $\boldsymbol{x}_c$ and noisy outputs $\boldsymbol{y}_c$. The goal of my synthesis algorithm is to synthesize a program which *best-fits* the noisy dataset $(\boldsymbol{x}_c, \boldsymbol{y}_c)$.

Input Loss:

Given a noisy dataset $\mathcal{D} = (\boldsymbol{x}_c, \boldsymbol{y}_c)$ of size n, to synthesize the *best-fit* program, the algorithm aims to synthesize the *best-fit* inputs to measure how well a program fits the given dataset. I use an input loss function $\mathcal{L}(\boldsymbol{x}, \boldsymbol{x}_c)$ to measure how incorrect our predicted inputs $\boldsymbol{x} \in X^n$ is with respect to the given dataset. Here X is the set of all possible inputs. For $\boldsymbol{x} \notin X^n$, I assume $\mathcal{L}(\boldsymbol{x}, \boldsymbol{x}_c) = \infty$. I also assume the input loss function $\mathcal{L}$ satisfies the following property:

$$\forall\ n \in \mathbb{N}.\ \forall\ \boldsymbol{x} \in X^n. \forall \delta \geq 0. \left| \left\{ \boldsymbol{x} \mid \boldsymbol{x} \in X^n, \mathcal{L}(\boldsymbol{x}, \boldsymbol{x}_c) \leq \delta \right\} \right| < \infty$$

i.e., if for all $n \in \mathbb{N}$, input vector $\boldsymbol{x}_c \in X^n$, $\delta \geq 0$, there exists only a finite number of input vectors $\boldsymbol{x} \in X^n$ with loss less than equal to δ.

I use the notation $\mathcal{L}_o$ to denote the *output* loss function and $\mathcal{L}_i$ to denote the *input* loss function.

Objective Function:

Given a noisy dataset $\mathcal{D} = (\boldsymbol{x}_c, \boldsymbol{y}_c)$, an output loss function $\mathcal{L}_o$, an input loss function

$\mathcal{L}_i$, and a complexity measure C, we modify the objective function U to integrate the input loss. Given a program p and noise-free inputs $\boldsymbol{x}$, we use the notation $U(\mathcal{L}_o(p[\boldsymbol{x}], \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{x}, \boldsymbol{x}_c), C(p))$ to denote the objective function value. Within this framework, **I assume a uniform regularizer**. For simplicity, I have removed the regularizer from the signature of the objective function.

I assume for any finite objective function value, both input loss and output loss is finite, i.e., for all δ, there exists a δ_I and δ_o, such that,

$$U(\mathcal{L}_o(p[\boldsymbol{x}], \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{x}, \boldsymbol{x}_c), C(p)) \leq \delta \implies \mathcal{L}_o(p[\boldsymbol{x}], \boldsymbol{y}_c) \leq \delta_o \text{ and } \mathcal{L}_i(\boldsymbol{x}, \boldsymbol{x}_c) \leq \delta_I$$

I assume both input and output loss function satisfy the following constraints:

$$\mathcal{L}_i(\langle x'_1, \ldots x'_n \rangle, \langle x_1, \ldots x_n \rangle) = \sum_{j=1}^{n} \mathcal{L}_i(x'_j, x_j)$$

$$\mathcal{L}_o(\langle z_1, \ldots z_n \rangle, \langle y_1, \ldots y_n \rangle) = \sum_{i=1}^{n} \mathcal{L}_o(z_i, y_i)$$

Optimization Problem:

Given a set of programs G, dataset $\mathcal{D} = (\boldsymbol{x}_c, \boldsymbol{y}_c)$ of size n, with both input and output corruptions, space of inputs X, an output loss function $\mathcal{L}_o$, an input loss function $\mathcal{L}_i$, a complexity measure C, an objective function U, a program p^* and inputs $\boldsymbol{x}^*$ *best-fit* the noisy dataset, if p^* and $\boldsymbol{x}^*$ minimizes the objective function, i.e.,

$$p^*, \boldsymbol{x}^* \in \underset{p \in G, \boldsymbol{x} \in X^n}{\arg\min}\, U(\mathcal{L}_o(p[\boldsymbol{x}], \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{x}, \boldsymbol{x}_c), C(p))$$

ϵ-Correctness:

Similar to my ϵ-correctness criterion (Definition 18), I relax the requirement to synthesize a program which is *close* to the optimal program, i.e., its objective function value is at most $\epsilon \geq 0$ greater than the minimum possible objective function value for any program in G. Formally, p^* is ϵ-correct, if and only if,

$$\min_{\boldsymbol{x} \in X^n} U(\mathcal{L}_o(p^*[\boldsymbol{x}], \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{x}, \boldsymbol{x}_c), C(p^*)) - \min_{p \in G, \boldsymbol{x} \in X^n} U(\mathcal{L}_o(p[\boldsymbol{x}], \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{x}, \boldsymbol{x}_c), C(p)) \leq \epsilon$$

$$\frac{t \in T_C}{[\![t]\!]^{\mathcal{P}}\phi \Rightarrow \alpha^{\mathcal{P}}(t = [\![t]\!]\phi)}\ (\textsc{Constant}) \qquad \frac{}{[\![x]\!]^{\mathcal{P}}\phi \Rightarrow \phi}\ (\textsc{Variable})$$

$$\frac{[\![e_1]\!]^{\mathcal{P}}\phi \Rightarrow \varphi_1 \quad [\![e_2]\!]^{\mathcal{P}}\phi \Rightarrow \varphi_2 \quad \dots \quad [\![e_k]\!]^{\mathcal{P}}\phi \Rightarrow \varphi_k}{[\![f(e_1, e_2, \dots e_k)]\!]^{\mathcal{P}}\phi \Rightarrow \alpha^{\mathcal{P}}([\![f(\varphi_1, \varphi_2, \dots \varphi_k)]\!]^{\#})}\ (\textsc{Function})$$

Figure 8-1: Abstract execution semantics for program p.

Note that, when $\epsilon = 0$, p^* minimizes the objective function.

Input Partitions:

Given the space of all possible input values X, a vector of m abstract values $\boldsymbol{\phi} = \langle \phi_1, \dots \phi_m \rangle$ are input partitions, if and only if,

$$X \subseteq \bigcup_{i=1}^{m} \gamma(\phi_i)$$

i.e., the input space is a subset of the set of values represented by the combined vector of abstract values.

Abstract Execution Semantics:

Given a program p, predicates $\mathcal{P}$, and partition ϕ, $[\![p]\!]^{\mathcal{P}}\phi$ denotes the abstract value of program p on abstract input ϕ, if the intermediate values are only represented via predicates in $\mathcal{P}$. Figure 8-1 presents the rules for computing $[\![p]\!]^{\mathcal{P}}\phi$. Given a vector of partitions $\boldsymbol{\phi} = \langle \phi_1, \dots \phi_m \rangle$, $[\![p]\!]^{\mathcal{P}}\boldsymbol{\phi}$ denotes the vector $\boldsymbol{\varphi} = \langle [\![p]\!]^{\mathcal{P}}\phi_1, \dots [\![p]\!]^{\mathcal{P}}\phi_n \rangle$.

I assume if a predicate $\psi \in \mathcal{U}$, then $\neg\psi \in \mathcal{U}$, i.e, if a predicate is present in the universe of predicates, then its negation also exists within the universe of predicates.

Abstract Objective Function Value:

Given a dataset $\mathcal{D} = (\boldsymbol{x}_c, \boldsymbol{y}_c)$ of size n, program p, m input partitions $\langle \phi_1, \dots \phi_m \rangle$, predicates $\mathcal{P}$, objective function U, output loss function $\mathcal{L}_o$, input loss function $\mathcal{L}_i$, and complexity measure C, I define the *abstract objective function* as:

$$u^* = \min_{\boldsymbol{\phi} \in \Phi} U(\mathcal{L}_o([\![p]\!]^{\mathcal{P}}\boldsymbol{\phi}, \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{\phi}, \boldsymbol{x}_c), C(p))$$

where $\Phi = \left\{ \langle \phi_{i_1}, \dots \phi_{i_n} \rangle \mid i_1, \dots i_n \in [1, m] \right\}$. Φ is the set of all possible n sized

permutations (with repetitions) of given input partitions.

Theorem 31. *Given a dataset $\mathcal{D} = (\boldsymbol{x}_c, \boldsymbol{y}_c)$ of size n, program p, m input partitions $\langle \phi_1, \dots \phi_m \rangle$, predicates $\mathcal{P}$, objective function U, output loss function $\mathcal{L}_o$, input loss function $\mathcal{L}_i$, complexity measure C, and space of inputs X,*

$$u^* = \min_{\boldsymbol{\phi} \in \Phi} U(\mathcal{L}_o([\![p]\!]^{\mathcal{P}} \boldsymbol{\phi}, \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{\phi}, \boldsymbol{x}_c), C(p))$$

where $\Phi = \left\{ \langle \phi_{i_1}, \dots \phi_{i_n} \rangle \mid i_1, \dots i_n \in [1, m] \right\}$, *then*

$$u^* \leq \min_{\boldsymbol{x} \in X^n} U(\mathcal{L}_o(p[\boldsymbol{x}], \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{x}, \boldsymbol{x}_c), C(p))$$

Proof.

$$u^* = \min_{\boldsymbol{\phi} \in \Phi} U(\mathcal{L}_o([\![p]\!]^{\mathcal{P}} \boldsymbol{\phi}, \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{\phi}, \boldsymbol{x}_c), C(p))$$

$$\leq \min_{\langle \phi_1^*, \dots \phi_n^* \rangle \in \Phi} \min_{x_i \in \gamma(\phi_i^*), i \in [1,n]} U(\mathcal{L}_o([\![p]\!]^{\mathcal{P}} \langle x_1, \dots x_n \rangle, \boldsymbol{y}_c), \mathcal{L}_i(\langle x_1, \dots x_n \rangle, \boldsymbol{x}_c), C(p))$$

$$\leq \min_{\langle \phi_1^*, \dots \phi_n^* \rangle \in \Phi} \min_{x_i \in \gamma(\phi_i^*), i \in [1,n]} U(\mathcal{L}_o(p[\langle x_1, \dots x_n \rangle], \boldsymbol{y}_c), \mathcal{L}_i(\langle x_1, \dots x_n \rangle, \boldsymbol{x}_c), C(p))$$

From definition of Φ,

$$= \min_{\phi_i^* \in \langle \phi_1, \dots \phi_m \rangle, i \in [1,n]} \min_{x_i \in \gamma(\phi_i^*), i \in [1,n]} U(\mathcal{L}_o(p[\langle x_1, \dots x_n \rangle], \boldsymbol{y}_c), \mathcal{L}_i(\langle x_1, \dots x_n \rangle, \boldsymbol{x}_c), C(p))$$

$\min_{A \in \mathcal{A}} \min_{x \in A} f(x) = \min_{x \in \bigcup_{A \in \mathcal{A}} A} f(x)$,

$$\leq \min_{x_i \in \bigcup_{j=1}^m \gamma(\phi_j), i \in [1,n]} U(\mathcal{L}_o(p[\langle x_1, \dots x_n \rangle], \boldsymbol{y}_c), \mathcal{L}_i(\langle x_1, \dots x_n \rangle, \boldsymbol{x}_c), C(p))$$

Since $X \subseteq \bigcup_{j=1}^m \gamma(\phi_j)$ and input loss (and therefore objective function) is infinite for $x \notin X$,

$$= \min_{x_i \in X, i \in [1,n]} U(\mathcal{L}_o(p[\langle x_1, \dots x_n \rangle], \boldsymbol{y}_c), \mathcal{L}_i(\langle x_1, \dots x_n \rangle, \boldsymbol{x}_c), C(p))$$

$$= \min_{\boldsymbol{x} \in X^n} U(\mathcal{L}_o(p[\boldsymbol{x}], \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{x}, \boldsymbol{x}_c), C(p))$$

Hence proved. □

8.2 Synthesis Algorithm

Figure 8-2 presents my synthesis algorithm which synthesizes the program p^* satisfying the ϵ correctness criterion. The Synthesize procedure takes a noisy dataset $\mathcal{D}$, a DSL G, a threshold $\epsilon \geq 0$, initial predicates $\mathcal{P}$, a universe of possible predicates $\mathcal{U}$, objective function U, output loss function $\mathcal{L}_o$, input loss function $\mathcal{L}_i$, and complexity measure C. I assume that $\mathsf{true}, \mathsf{false} \in \mathcal{P}$. All the procedures and sub-procedures are parameterized by an objective function, an output loss function, an input loss function, and a complexity measure. I remove these parameters from the signature of Synthesize (and other methods) for simplicity.

The synthesis algorithm consists of a refinement loop (line 2-8). Given predicates $\mathcal{P}$, the algorithm first partitions the input space to construct a vector of abstract values $\boldsymbol{\phi}$. As the synthesis algorithm proceeds the synthesis algorithm will increase the set of predicates $\mathcal{P}$ and uses these predicates to further partition the space of inputs.

The algorithm then constructs an abstract finite tree automaton (line 4) with the current partitions $\boldsymbol{\phi}$ and current set of predicates $\mathcal{P}$ using rules presented in Figure 8-4. The algorithm then uses the MinCost function to generate a candidate program p^* and candidate inputs $\boldsymbol{x}^*$ (line 5). p^* minimizes the abstract objective function and u^* is the abstract objective function value of program p^*.

The algorithm then computes the distance between the concrete objective function value of p^* over inputs $\boldsymbol{x}^*$ and the abstract objective function value u^*. If this distance is not greater than ϵ, the algorithm terminates and returns p^* (line 6).

Otherwise the algorithm augments the set of predicates to $\mathcal{P}$ to refine the AFTA and further partition the input space. To accomplish this, the algorithm first constructs an input-output example (x, y) using the noisy dataset $\mathcal{D}$ and input partitions $\boldsymbol{\phi}$. Given a noisy input x^* in the dataset and corresponding noisy output y, x is a candidate noise-free input corresponding to x^*.

The algorithm then uses the procedure OptimizeAndBackPropagate to expand the set of predicates (line 8). This procedure is identical to the OptimizeAndBackPropagate procedure presented in Chapter 6 (Algorithm 6-5).

1: **procedure** SYNTHESIZE($\mathcal{D}, G, \epsilon, \mathcal{P}, \mathcal{U}$)
input: noisy dataset $\mathcal{D} = (\boldsymbol{x}_c, \boldsymbol{y}_c)$, DSL G, and tolerance ϵ.
input: initial predicates $\mathcal{P}$, and universe of predicates $\mathcal{U}$.
output: A program p^*, such that, p^* satisfies the *ϵ-correctness* condition.
2: **while** true **do**
3: $\boldsymbol{\phi} = \mathsf{PartitionSpace}(\mathcal{P})$;
4: $\mathcal{A} := \mathsf{ConstructAFTA}(\boldsymbol{\phi}, G, \mathcal{P})$;
5: $p^*, \boldsymbol{x}^*, u^* := \mathsf{MinCost}(\mathcal{A}, \boldsymbol{\phi}, \mathcal{D})$;
6: **if** $\mathsf{Distance}(p^*, \boldsymbol{x}^*, u^*, \mathcal{D}, \mathcal{P}) \leq \epsilon$ **then return** p^*;
7: $x, y := \mathsf{PickDimension}(p^*, \boldsymbol{x}^*, \mathcal{D}, \mathcal{P})$;
8: $\mathcal{P} := \mathcal{P} \bigcup \mathsf{OptimizeAndBackPropagate}(p^*, x, y, \mathcal{P}, \mathcal{U})$;

Figure 8-2: Algorithm for noisy program synthesis with input corruptions.

1: **procedure** DIVIDESPACE($\varphi, \mathcal{P}$)
2: **if** $\mathcal{P} = \emptyset$ **then**
3: **return** $\{\varphi\}$;
4: $\psi \in \mathcal{P}$;
5: $S_p := \mathsf{DivideSpace}(\varphi \wedge \psi, \mathcal{P} - \{\psi\})$;
6: $S_n := \mathsf{DivideSpace}(\varphi \wedge \neg\psi, \mathcal{P} - \{\psi\})$;
7: **return** $S_p \cup S_n$;
8: **procedure** PARTITIONSPACE($\mathcal{P}$)
input: Predicates $\mathcal{P}$.
output: Partitions the input space into subspaces based on predicates $\mathcal{P}$.
9: **return** $\mathsf{DivideSpace}(\mathsf{true}, \mathcal{P})$;

Figure 8-3: Algorithm for partitioning the space of inputs into subspaces based on predicates $\mathcal{P}$.

I discuss each of these sub-procedures in detail next.

8.2.1 Creating Input Partitions

Figure 8-3 presents the procedure to partition the input space, given predicates $\mathcal{P}$. The algorithm uses all predicates in $\mathcal{P}$ to construct these partitions. For each partition ϕ and each predicate $\psi \in \mathcal{P}$, either $\psi \implies \phi$ or $\neg\psi \implies \phi$. The procedure recursively computes the following set of partitions:

$$\boldsymbol{\phi} = \Big\{\varphi_1 \wedge \ldots \varphi_k \mid \varphi_1 \in \{\psi_1, \neg\psi_1\}, \ldots \varphi_k \in \{\psi_k, \neg\psi_k\}, \mathcal{P} = \{\psi_1, \ldots \psi_k\}\Big\}$$

Theorem 32. *Given predicates $\mathcal{P}$ and input space X, if $\langle \phi_1, \ldots \phi_m \rangle := \mathsf{PartitionSpace}(\mathcal{P})$,*

then

$$X \subseteq \bigcup_{i=1}^{m} \gamma(\phi_i)$$

Proof. Consider the function DivideSpace, if $S = \mathsf{DivideSpace}(\varphi, \mathcal{P})$, then

$$\gamma(\varphi) = \bigcup_{\phi \in S} \gamma(\phi)$$

Proof by induction over size of the set of predicates $\mathcal{P}$.

Base Case: If $\mathcal{P} = \emptyset$, then $\{\varphi\} = \mathsf{DivideSpace}(\varphi, \emptyset)$.

$$\gamma(\varphi) = \bigcup_{\phi \in \{\varphi\}} \gamma(\phi)$$

Induction Hypothesis: If $\mathcal{P}$ contains n predicates and $S = \mathsf{DivideSpace}(\varphi, \mathcal{P})$, then

$$\gamma(\varphi) = \bigcup_{\phi \in S} \gamma(\phi)$$

Induction Case: Let $\mathcal{P}$ contain $n+1$ predicates and $S = \mathsf{DivideSpace}(\varphi, \mathcal{P})$. Let ψ be a predicate in $\mathcal{P}$. Let $S_p = \mathsf{DivideSpace}(\varphi \wedge \psi, \mathcal{P} - \{\psi\})$, then

$$\gamma(\varphi \wedge \psi) = \bigcup_{\phi \in S_p} \gamma(\phi)$$

Let $S_n = \mathsf{DivideSpace}(\varphi \wedge \neg\psi, \mathcal{P} - \{\psi\})$, then

$$\gamma(\varphi \wedge \neg\psi) = \bigcup_{\phi \in S_n} \gamma(\phi)$$

Since $S = S_p \cup S_n$,

$$\bigcup_{\phi \in S} \gamma(\phi) = \Big(\bigcup_{\phi \in S_p} \gamma(\phi) \Big) \cup \Big(\bigcup_{\phi \in S_n} \gamma(\phi) \Big) = \gamma(\varphi \wedge \psi) \cup \gamma(\varphi \wedge \neg\psi) = \gamma(\varphi)$$

Therefore, using induction, $S = \mathsf{DivideSpace}(\varphi, \mathcal{P})$, then

$$\gamma(\varphi) = \bigcup_{\phi \in S} \gamma(\phi)$$

Since $X \subseteq \gamma(\top)$, if $\langle \phi_1, \ldots \phi_m \rangle = \mathsf{PartitionSpace}(\mathcal{P}) = \mathsf{DivideSpace}(\top, \mathcal{P})$,

$$X \subseteq \bigcup_{i=1}^{m} \gamma(\phi_i)$$

□

Theorem 33. *Given a set of predicates $\mathcal{P}$, if $\langle \phi_1, \ldots \phi_m \rangle := \mathsf{PartitionSpace}(\mathcal{P})$, then for all $i = 1, \ldots m$, $\forall x_i \in \gamma(\phi_i).\alpha^{\mathcal{P}}(x = x_i) \iff \phi$.*

Proof. $\alpha^{\mathcal{P}}$ of $x = v$ is the strongest predicate containing predicates $\mathcal{P}$, $\phi_i \implies \alpha^{\mathcal{P}}(x = x_i) \implies (x = x_i)$. But ϕ contains all predicates in $\mathcal{P}$, therefore $\alpha^{\mathcal{P}}(x = x_i)$ cannot be stronger than ϕ_i, $\alpha^{\mathcal{P}}(x = x_i) \implies \phi_i$. Therefore, $\alpha^{\mathcal{P}}(x = x_i) \iff \phi_i$. □

Theorem 34. *Given a set of predicates $\mathcal{P}$, and program p, if $\langle \phi_1, \ldots \phi_m \rangle := \mathsf{PartitionSpace}(\mathcal{P})$, then for all $i = 1, \ldots m$, $\forall x_i \in \gamma(\phi_i).[\![p]\!]^{\mathcal{P}} x_i = [\![p]\!]^{\mathcal{P}} \phi_i$.*

Proof. Let $x_i \in \gamma(\phi_i)$. Given a program p, $[\![p]\!]^{\mathcal{P}} x_i = [\![p]\!]^{\mathcal{P}} \phi_i$.
Proof using induction over size of program p,
Base Case: Consider programs p of size 1.
Case 1: p is a constant t. From *Const* rule for computing $[\![t]\!]^{\mathcal{P}} x_i$ (Figure 6-1) and $[\![t]\!]^{\mathcal{P}} \phi_i$ (Figure 8-1), $[\![t]\!]^{\mathcal{P}} x_i = [\![t]\!]^{\mathcal{P}} \phi_i$.
Case 2: p is a variable t. From *Var* rule for computing $[\![x]\!]^{\mathcal{P}} x_i$ (Figure 6-1) and $[\![x]\!]^{\mathcal{P}} \phi_i$ (Figure 8-1) and Theorem 33, $[\![x]\!]^{\mathcal{P}} x_i = [\![x]\!]^{\mathcal{P}} \phi_i$.
Induction Hypothesis: Given a program p of size less than n, $[\![p]\!]^{\mathcal{P}} x_i = [\![p]\!]^{\mathcal{P}} \phi_i$.
Induction Step: Given a program p of size n. $p = f(e_1, \ldots e_k)$. Size of $e_1, \ldots e_k$ is less than n, therefore, $l = 1, \ldots k$, $[\![e_l]\!]^{\mathcal{P}} x_i = [\![e_l]\!]^{\mathcal{P}} \phi_i$. From *Function* rule for computing $[\![f(e_1, \ldots e_k)]\!]^{\mathcal{P}} x_i$ (Figure 6-1) and $[\![f(e_1, \ldots e_k)]\!]^{\mathcal{P}} \phi_i$ (Figure 8-1), $[\![e]\!]^{\mathcal{P}} x_i = [\![e]\!]^{\mathcal{P}} \phi_i$.
Hence proved. □

8.2.2 Abstract Finite Tree Automata

Given predicates $\mathcal{P}$, DSL G, and input partitions $\boldsymbol{\phi}$, Figure 8-4 presents the rules for constructing an AFTA (Q, Q_f, Δ). The structure of the AFTA is similar to the AFTA presented in my original abstraction refinement based synthesis algorithm

$$\frac{\boldsymbol{\varphi} = \langle \psi_1, \ldots \psi_m \rangle}{q_x^{\boldsymbol{\varphi}} \in Q} \text{ (VAR)} \quad \frac{q_{s_0}^{\boldsymbol{\varphi}} \in Q}{q_{s_0}^{\boldsymbol{\varphi}} \in Q_f} \text{ (FINAL)}$$

$$\frac{t \in T_C, \boldsymbol{\varphi} = \alpha^{\mathcal{P}}\Big(\langle t = [\![t]\!], \ldots t = [\![t]\!] \rangle\Big), |\boldsymbol{\varphi}| = m}{q_t^{\boldsymbol{\varphi}} \in Q} \text{ (CONST)}$$

$$\frac{\begin{array}{c} s \to f(s_1, \ldots, s_k) \in P, q_{s_1}^{\boldsymbol{\varphi}_1}, \ldots, q_{s_k}^{\boldsymbol{\varphi}_k} \in Q, \\ \varphi_j = \alpha^{\mathcal{P}}\Big([\![f(\varphi_{1j}, \ldots, \varphi_{kj})]\!]^{\#}\Big), \boldsymbol{\varphi} = \langle \varphi_1, \ldots \varphi_m \rangle \end{array}}{q_s^{\boldsymbol{\varphi}} \in Q, f(q_{s_1}^{\boldsymbol{\varphi}_1}, \ldots, q_{s_k}^{\boldsymbol{\varphi}_k}) \to q_s^{\boldsymbol{\varphi}} \in \Delta} \text{ (PROD)}$$

Figure 8-4: Rules for constructing FTA $\mathcal{A} = (Q, Q_f, \Delta)$ with abstract values, given abstract inputs $\boldsymbol{\psi} = \langle \psi_1, \ldots \psi_m \rangle$.

(Chapter 6). I modify the *Var* rule to handle input partitions $\boldsymbol{\phi}$. Instead of using $\alpha^{\mathcal{P}}(\boldsymbol{x}_c)$ as the abstract value, where $\boldsymbol{x}_c$ are the inputs in the noisy dataset, the modified method ConstructAFTA attaches the input partitions $\boldsymbol{\phi}$ to the AFTA state $q_x^{\boldsymbol{\phi}}$, associated with input symbol x.

8.2.3 Minimum Cost Candidate

I present the implementation of the procedure MinCost in Figure 8-5. Given AFTA (Q, Q_f, Δ), dataset $\mathcal{D} = (\boldsymbol{x}_c, \boldsymbol{y}_c)$, input partitions $\boldsymbol{\phi} = \langle \phi_1, \ldots \phi_m \rangle$, objective function U, output loss function $\mathcal{L}_o$, input loss function $\mathcal{L}_i$, and complexity measure C, MinCost returns a program p^*, candidate input vector $\boldsymbol{x}^*$, and a minimum objective function value u^*. Formally,

$$p^*, \langle \phi_1^*, \ldots \phi_n^* \rangle \in \underset{p^* \in G, \boldsymbol{\phi} \in \Phi}{\arg\min} U(\mathcal{L}_o([\![p]\!]^{\mathcal{P}} \boldsymbol{\phi}, \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{\phi}, \boldsymbol{x}_c), C(p))$$

$$\boldsymbol{x}^* = \langle x_1^*, \ldots x_n^* \rangle, \quad x_j^* \in \underset{x \in \gamma(\phi_j^*)}{\arg\min} \mathcal{L}_i(x, x_j), j \in [1, n]$$

where $\Phi = \Big\{ \langle \phi_{i_1}, \ldots \phi_{i_n} \rangle \mid i_1, \ldots i_n \in [1, m] \Big\}$, $\boldsymbol{x}_c = \langle x_1, \ldots x_n \rangle$, and

$$u^* = U(\mathcal{L}_o([\![p^*]\!]^{\mathcal{P}} \boldsymbol{\phi}^*, \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{\phi}^*, \boldsymbol{x}_c), C(p^*))$$

Theorem 35. *Given predicates $\mathcal{P}$, DSL G, dataset $\mathcal{D} = (\boldsymbol{x}_c, \boldsymbol{y}_c)$, objective function U,*

1: **procedure** MINCOST($\mathcal{A}, \boldsymbol{\phi}, \mathcal{D}$)
input: abstract inputs $\boldsymbol{\phi}$, AFTA $\mathcal{A} = (Q, Q_f, \Delta)$, and dataset $\mathcal{D} = (\boldsymbol{x}_c, \boldsymbol{y}_c)$.
output: A program p^* and inputs $\boldsymbol{x}^*$, which minimizes the abstract objective function.

2: $p_r = \mathsf{null}$; $u_r = \infty$; $n := |\boldsymbol{x}_c|$; $m := |\boldsymbol{\phi}|$; $\boldsymbol{x}_r := \mathsf{null}$;
3: **for** $q^{\boldsymbol{\varphi}}_{s_0} \in Q_f$ **do**
4: $p, c := \mathsf{LeastComplex}(q^{\boldsymbol{\varphi}}_{s_0}, \mathcal{A}, G)$;
▷ Least complex program for a given accepting state.
5: $\Phi := \left\{(\langle \phi_{i_1}, \dots \phi_{i_n} \rangle, \langle \varphi_{i_1}, \dots \varphi_{i_n} \rangle) \mid i_1, \dots i_n \in [1, m]\right\}$;
6: $(\boldsymbol{\phi}^*, \boldsymbol{\varphi}^*) := \underset{(\boldsymbol{\phi}^*, \boldsymbol{\varphi}^*) \in \Phi}{\arg\min}\, U(\mathcal{L}_o(\boldsymbol{\varphi}^*, \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{\phi}^*, \boldsymbol{x}_c), C(p))$;
7: $u := U(\mathcal{L}_o(\boldsymbol{\varphi}^*, \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{\phi}^*, \boldsymbol{x}_c), C(p))$;
8: **if** $u \leq u_r$ **then**
9: $p_r = p$; $u_r = u$;
10: **for** $j := 1 \dots n$ **do**
11: $x_j := \underset{x \in \gamma(\phi^*_j)}{\arg\min}\, \mathcal{L}_i(x, x_{c\ j})$;
12: $\boldsymbol{x}_r = \langle x_1, \dots x_n \rangle$;
13: **return** $p_r, \boldsymbol{x}_r, u_r$;

Figure 8-5: Procedure for synthesizing the program and inputs which minimizes the abstract objective function.

output loss function $\mathcal{L}_o$, input loss function $\mathcal{L}_i$, complexity measure C, input partitions $\boldsymbol{\phi}$, and FTA $\mathcal{A} = \mathsf{ConstructAFTA}(\boldsymbol{\phi}, G, \mathcal{P})$, if $p^*, \boldsymbol{x}^*, u^* = \mathsf{MinCost}(\mathcal{A}, \boldsymbol{\phi}, \mathcal{D})$ then

$$p^*, \boldsymbol{\phi}^* \in \underset{p^* \in G, \boldsymbol{\phi}^* \in \Phi^*}{\arg\min}\, U(\mathcal{L}_o([\![p]\!]^{\mathcal{P}} \boldsymbol{\phi}, \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{\phi}, \boldsymbol{x}_c), C(p))$$

where $\Phi^* = \left\{\langle \phi_{i_1}, \dots \phi_{i_n} \rangle \mid i_1, \dots i_n \in [1, m]\right\}$, $n = |\boldsymbol{x}_c|$, $m = |\boldsymbol{\phi}|$, *and*

$$\boldsymbol{x}^* = \langle x^*_1, \dots x^*_n \rangle, \quad x^*_j \in \underset{x \in \gamma(\phi^*_j)}{\arg\min}\, \mathcal{L}_i(x, x_{c\ j}), j \in [1, n]$$

$$u^* = U(\mathcal{L}_o([\![p^*]\!]^{\mathcal{P}} \boldsymbol{\phi}^*, \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{\phi}^*, \boldsymbol{x}_c), C(p^*)) = U(\mathcal{L}_o([\![p^*]\!]^{\mathcal{P}} \boldsymbol{x}^*, \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{x}^*, \boldsymbol{x}_c), C(p^*))$$

Also, if

$$U(\mathcal{L}_o(p^*[\boldsymbol{x}^*], \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{x}^*, \boldsymbol{x}_c), C(p^*)) - U(\mathcal{L}_o([\![p^*]\!]^{\mathcal{P}} \boldsymbol{x}^*, \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{x}^*, \boldsymbol{x}_c), C(p^*)) > 0$$

then

$$\mathcal{L}_o(p^*[\boldsymbol{x}^*], \boldsymbol{y}_c) - \mathcal{L}_o([\![p^*]\!]^{\mathcal{P}}\boldsymbol{x}^*, \boldsymbol{y}_c) > 0$$

Proof. Given a state $q^{\boldsymbol{\varphi}}_{s_0} \in Q_f$, from Theorem 2 and Theorem 33,

$$p, c = \mathsf{LeastComplex}(q^{\boldsymbol{\varphi}}, \mathcal{A}, G) \iff p \in \underset{p \in G[\boldsymbol{\phi} \to \boldsymbol{\varphi}]}{\arg\min}\, C(p)$$

where $G[\boldsymbol{\phi} \to \boldsymbol{\varphi}] = \{p \in G \mid [\![p]\!]^{\mathcal{P}}\boldsymbol{\phi} = \boldsymbol{\varphi}\}$. Note that,

$$\boldsymbol{\varphi}^* = [\![p^*]\!]^{\mathcal{P}}\boldsymbol{\phi}^*$$

I can rewrite the (line 6) as

$$\boldsymbol{\phi}^* = \underset{\boldsymbol{\phi}^* \in \Phi^*}{\arg\min}\, U(\mathcal{L}_o([\![p]\!]^{\mathcal{P}}\boldsymbol{\phi}^*, \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{\phi}^*, \boldsymbol{x}_c), C(p))$$

Therefore, u (line 7) is the abstract objective function value of program p. Note that,

$$p = \underset{p \in G[\boldsymbol{\phi} \to \boldsymbol{\varphi}]}{\arg\min} \underset{\boldsymbol{\phi}^* \in \Phi^*}{\min}\, U(\mathcal{L}_o([\![p]\!]^{\mathcal{P}}\boldsymbol{\phi}^*, \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{\phi}^*, \boldsymbol{x}_c), C(p))$$

The algorithm iterates through all states in Q_f and maintains the program p_r with the minimum abstract objective function value. Therefore,

$$q^{\boldsymbol{\varphi}}_{s_0}, p_r = \underset{q^{\boldsymbol{\varphi}}_{s_0} \in Q_f, p \in G[\boldsymbol{\phi} \to \boldsymbol{\varphi}]}{\arg\min} \underset{\boldsymbol{\phi}^* \in \Phi^*}{\min}\, U(\mathcal{L}_o([\![p]\!]^{\mathcal{P}}\boldsymbol{\phi}^*, \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{\phi}^*, \boldsymbol{x}_c), C(p))$$

$$p_r = \underset{p \in \bigcup_{q^{\boldsymbol{\varphi}}_{s_0} \in Q_f} G[\boldsymbol{\phi} \to \boldsymbol{\varphi}]}{\arg\min} \underset{\boldsymbol{\phi}^* \in \Phi^*}{\min}\, U(\mathcal{L}_o([\![p]\!]^{\mathcal{P}}\boldsymbol{\phi}^*, \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{\phi}^*, \boldsymbol{x}_c), C(p))$$

From Theorem 21, $\bigcup_{q^{\boldsymbol{\varphi}}_{s_0} \in Q_f} G[\boldsymbol{\phi} \to \boldsymbol{\varphi}] = G$. Therefore,

$$p_r = \underset{p \in G}{\arg\min} \underset{\boldsymbol{\phi}^* \in \Phi^*}{\min}\, U(\mathcal{L}_o([\![p]\!]^{\mathcal{P}}\boldsymbol{\phi}^*, \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{\phi}^*, \boldsymbol{x}_c), C(p))$$

i.e., p_r minimizes the abstract objective function value.

$$\boldsymbol{\phi}^* = \underset{\boldsymbol{\phi}^* \in \Phi^*}{\arg\min} \underset{p \in G}{\min}\, U(\mathcal{L}_o([\![p]\!]^{\mathcal{P}}\boldsymbol{\phi}^*, \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{\phi}^*, \boldsymbol{x}_c), C(p))$$

From line 12 and 13,

$$\boldsymbol{x}^* = \langle x_1^*, \ldots x_n^* \rangle, \quad x_j^* \in \underset{x \in \gamma(\phi_j^*)}{\arg\min}\, \mathcal{L}_i(x, x_{c\ i}), j \in [1, n]$$

Since $\boldsymbol{\varphi}^* = [\![p^*]\!]^{\mathcal{P}} \boldsymbol{\phi^*}$, from line 7 and line 9:

$$u^* = U(\mathcal{L}_o([\![p^*]\!]^{\mathcal{P}} \boldsymbol{\phi}^*, \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{\phi}^*, \boldsymbol{x}_c), C(p^*))$$

From Theorem 33 and definition of $\boldsymbol{x}^*$,

$$u^* = U(\mathcal{L}_o([\![p^*]\!]^{\mathcal{P}} \boldsymbol{x}^*, \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{x}^*, \boldsymbol{x}_c), C(p^*))$$

Therefore, if

$$U(\mathcal{L}_o(p^*[\boldsymbol{x}^*], \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{x}^*, \boldsymbol{x}_c), C(p^*)) - U(\mathcal{L}_o([\![p^*]\!]^{\mathcal{P}} \boldsymbol{x}^*, \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{x}^*, \boldsymbol{x}_c), C(p^*)) > 0$$

then

$$\mathcal{L}_o(p^*[\boldsymbol{x}^*], \boldsymbol{y}_c) - \mathcal{L}_o([\![p^*]\!]^{\mathcal{P}} \boldsymbol{x}^*, \boldsymbol{y}_c) > 0$$

□

8.2.4 Termination Condition and Tolerance

Given a candidate program p^*, candidate noise-free inputs $\boldsymbol{x}^*$, predicates $\mathcal{P}$, dataset $\mathcal{D} = (\boldsymbol{x}_c, \boldsymbol{y}_c)$, objective function U, output loss function $\mathcal{L}_o$, input loss function $\mathcal{L}_i$, and complexity measure C, the Distance function returns the difference between the concrete objective function value of p^* and candidate noise-free inputs $\boldsymbol{x}^*$ over noisy dataset $\mathcal{D}$ and the abstract objective function value over noisy dataset $\mathcal{D}$. Formally:

$$\mathsf{Distance}(p^*, \boldsymbol{x}^*, u^*, (\boldsymbol{x}_c, \boldsymbol{y}_c), \mathcal{P}) := U(\mathcal{L}_o(p^*[\boldsymbol{x}^*], \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{x}^*, \boldsymbol{x}_c), C(p)) - u^*$$

The algorithm terminates if the distance is less than or equal to the tolerance level ϵ. Since

$$u^* \leq \min_{p \in G, \boldsymbol{x} \in X^n} U(\mathcal{L}_o(p[\boldsymbol{x}], \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{x}, \boldsymbol{x}_c), C(p))$$

the algorithm will only terminate if the concrete objective function value of if program p^* and noise-free inputs $\boldsymbol{x}^*$ is not more than ϵ larger than the minimum concrete function value for all programs in the DSL G and noise-free inputs inputs in X^n.

Theorem 36. (Soundness) *Given a dataset $\mathcal{D}$, DSL G, tolerance $\epsilon \geq 0$, universe of predicates $\mathcal{U}$, initial predicates $\mathcal{P}$, objective function U, output loss function $\mathcal{L}_o$, input loss function $\mathcal{L}_i$, and the complexity measure C, if Algorithm 8-2 returns the program p^*, then*

$$\min_{\boldsymbol{x} \in X^n} U(\mathcal{L}_o(p^*[\boldsymbol{x}], \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{x}, \boldsymbol{x}_c), C(p^*)) - \min_{p \in G, \boldsymbol{x} \in X^n} U(\mathcal{L}_o(p[\boldsymbol{x}], \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{x}, \boldsymbol{x}_c), C(p)) \leq \epsilon$$

Proof. Let us assume that the algorithm terminates on the i^{th} iteration. Let $\mathcal{P}^*$ and $\boldsymbol{x}^*$ be the set of predicates and noise-free inputs when the algorithm terminates. Note that the algorithm terminates when:

$$\epsilon \geq U(\mathcal{L}_o(p^*[\boldsymbol{x}^*], \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{x}^*, \boldsymbol{x}_c), C(p^*)) - u^* \geq \min_{\boldsymbol{x} \in X^n} U(\mathcal{L}_o(p^*[\boldsymbol{x}], \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{x}, \boldsymbol{x}_c), C(p^*)) - u^*$$

From Theorem 31 and Theorem 35,

$$\min_{\boldsymbol{x} \in X^n} U(\mathcal{L}_o(p^*[\boldsymbol{x}], \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{x}, \boldsymbol{x}_c), C(p^*)) - \min_{p \in G, \boldsymbol{x} \in X^n} U(\mathcal{L}_o(p[\boldsymbol{x}], \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{x}, \boldsymbol{x}_c), C(p)) \leq \epsilon$$

Hence proved. □

8.2.5 Abstraction Refinement Based Optimization

Given a dataset $\mathcal{D}$, and predicates $\mathcal{P}$, the program p^* and noise-free inputs $\boldsymbol{x}^*$ minimize the abstract objective function. If the algorithm did not terminate then $\mathsf{Distance}(p^*, \boldsymbol{x}^*, u^*, \mathcal{D}, \mathcal{P}) > \epsilon$. Just using predicates $\mathcal{P}$ we cannot prove p^* is ϵ-correct. Therefore, in order to find the optimal program, we need to expand the set of predi-

cates $\mathcal{P}$. Since $\mathsf{Distance}(p^*, \boldsymbol{x}^*, u^*, \mathcal{D}, \mathcal{P}) > 0$, from Theorem 35:

$$\mathcal{L}_o(p^*[\boldsymbol{x}^*], \boldsymbol{y}_c) - \mathcal{L}_o([\![p^*]\!]^{\mathcal{P}} \boldsymbol{x}^*, \boldsymbol{y}_c) > 0$$

The algorithm expands $\mathcal{P}$ to $\mathcal{P}'$ to reduce this difference between the abstract loss function and the concrete loss function, i.e.,

$$\mathcal{L}_o(p^*[\boldsymbol{x}^*], \boldsymbol{y}_c) - \mathcal{L}_o([\![p^*]\!]^{\mathcal{P}} \boldsymbol{x}^*, \boldsymbol{y}_c) > \mathcal{L}_o(p^*[\boldsymbol{x}^*], \boldsymbol{y}_c) - \mathcal{L}_o([\![p^*]\!]^{\mathcal{P}'} \boldsymbol{x}^*, \boldsymbol{y}_c) \geq 0$$

Given $\boldsymbol{x}^* = \langle x_1^*, \ldots x_n^* \rangle$ and $\boldsymbol{y}_c = \langle y_1, \ldots y_n \rangle$, the synthesis algorithm achieves this by select an input-output example (x_i^*, y_i), such that,

$$\mathcal{L}_o(p^*[x_i^*], y_i) - \mathcal{L}_o([\![p^*]\!]^{\mathcal{P}} x_i^*, y_i) > 0$$

i.e., the concrete output loss of this example is greater than the objective output loss over this example. The synthesis algorithm uses a pluggable procedure $\mathsf{PickDimension}$ to select this example. The algorithm allows us to plug any implementation of the procedure $\mathsf{PickDimension}$, assuming it satisfies the following constraint:

$$(x_i^*, y_i) = \mathsf{PickDimension}(p^*, \langle x_1^*, \ldots x_n^* \rangle, (\langle x_1, \ldots x_n \rangle, \langle y_1, \ldots y_n \rangle), \mathcal{P}) \implies$$

$$\mathcal{L}_o(p^*[x_i^*], y_i) - \mathcal{L}_o([\![p^*]\!]^{\mathcal{P}} x_i^*, y_i) > 0$$

Given this input-output example (x_i^*, y_i), $\mathsf{OptimizeAndBackPropagate}$ expands the set of predicates $\mathcal{P}$ to $\mathcal{P}'$, such that:

$$\mathcal{L}_o([\![p^*]\!]^{\mathcal{P}} x_i^*, y_i) < \mathcal{L}_o([\![p^*]\!]^{\mathcal{P}'} x_i^*, y_i) \leq \mathcal{L}_o(p[x_i^*], y_i)$$

This expanded set of predicates allows us to improve our estimation of the abstract output loss function for the candidate program (and potentially other programs).

The synthesis algorithm uses the $\mathsf{OptimizeAndBackPropagate}$ procedure presented in Chapter 6 (Figure 6-5). The procedure synthesizes the strongest formula ψ^*, such

that, $(s_0 = p^*[x_i^*]) \implies \psi^*$ and:

$$\mathcal{L}_o([\![p^*]\!]^{\mathcal{P}} x_i^*, y_i) < \mathcal{L}_o(([\![p^*]\!]^{\mathcal{P}} x_i^*) \wedge \psi^*, y_i) \leq \mathcal{L}_o(p[x_i^*], y_i)$$

Theorem 37. *Given a dataset $\mathcal{D} = (\boldsymbol{x}_c, \boldsymbol{y}_c)$, a DSL G, tolerance $\epsilon \geq 0$, universe of predicates $\mathcal{U}$, initial predicates $\mathcal{P}$, objective function U, output loss function $\mathcal{L}_o$, input loss function $\mathcal{L}_i$, and the complexity measure C, Algorithm 8-2 will eventually return some program p^*.*

Proof.

$$\delta_u = \min_{p \in G, \boldsymbol{x} \in X^n} U(\mathcal{L}_o(p[\boldsymbol{x}], \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{x}, \boldsymbol{x}_c), C(p))$$

Let A_i be the FTA constructed in the i^{th} iteration of Algorithm 8-2. Let $\mathcal{P}_i$ be the set of predicates, $p_i, \boldsymbol{x}_i$, and u_i be the program, noise-free inputs, abstract objective function value returned by MinCost in the i^{th} iteration. Let x_i^*, y_i be the noise-free input and the noisy output returned by PickDimension in the i^{th} iteration. Note that, in each iteration $u_i \leq \delta_u$ (the abstract loss function is less than the minimum concrete loss function). This implies, there exists δ_I and δ_o, such that,

$$\mathcal{L}_o([\![p_i]\!]^{\mathcal{P}_i} \boldsymbol{x}_i, \boldsymbol{x}_c) \leq \delta_o \text{ and } \mathcal{L}_i(\boldsymbol{x}_i, \boldsymbol{x}_c) \leq \delta_I$$

There exists only a finite set of input $\mathcal{X}$, such that, $\forall \boldsymbol{x} \notin \mathcal{X}. \mathcal{L}_i(\boldsymbol{x}, \boldsymbol{x}_c) > \delta_I$.

Using Theorem 25, if $\mathsf{Distance}(p_i, \boldsymbol{x}_i^*, \mathcal{D}, \mathcal{P}_i) > \epsilon$, then for all $k > i$:

$$\min(\mathcal{L}_o(p_i[\boldsymbol{x}_i], \boldsymbol{y}_c), \mathcal{L}_o([\![p_i]\!]^{\mathcal{P}_i} \boldsymbol{x}_i, \boldsymbol{y}_c) + \delta) \leq \mathcal{L}_o([\![p_i]\!]^{\mathcal{P}_k} \boldsymbol{x}_i, \boldsymbol{y}_c) \leq \mathcal{L}_o(p_i[\boldsymbol{x}_i], \boldsymbol{y}_c)$$

In each iteration, the algorithm picks a program $p \in G$, picks an input $\boldsymbol{x}$ from a finite set $\mathcal{X}$ and increases the abstract loss value.

Also note that,

$$\mathcal{L}_o([\![p_i]\!]^{\mathcal{P}_k} \boldsymbol{x}_i, \boldsymbol{y}_c) \leq \delta_o + \delta$$

because if $\mathcal{L}_o([\![p_i]\!]^{\mathcal{P}_k} \boldsymbol{x}_i, \boldsymbol{y}_c) > \delta_o$, then the abstract objective function value of p_i and $\boldsymbol{x}_i$ is greater than δ_u. In this case, MinCost will never return p_i and $\boldsymbol{x}_i$.

Therefore, in the worst case, MinCost will return any combination of a program p and an input $\boldsymbol{x}$ at most $\frac{\delta_o}{\delta} + 1$ times.

The space of programs is finite (due to the restriction of the size of the AFTA) and the space of potential inputs with input loss less than the bound δ_I is finite. Each combination of a program and an input can only be candidates a finite number of times. Therefore, the algorithm will terminate in finite number of iterations.

□

Theorem 38. *Given a dataset $\mathcal{D}$, tolerance $\epsilon \geq 0$, universe of predicates $\mathcal{U}$, initial predicates $\mathcal{P}$, objective function U, output loss function $\mathcal{L}_o$, input loss function $\mathcal{L}_i$, and the complexity measure C, the Algorithm 8-2 will return a program p^*, such that,*

$$\min_{\boldsymbol{x} \in X^n} U(\mathcal{L}_o(p^*[\boldsymbol{x}], \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{x}, \boldsymbol{x}_c), C(p^*)) - \min_{p \in G, \boldsymbol{x} \in X^n} U(\mathcal{L}_o(p[\boldsymbol{x}], \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{x}, \boldsymbol{x}_c), C(p)) \leq \epsilon$$

Proof. From Theorem 37, Algorithm 8-2 will eventually terminate and return a program p^*. From Theorem 36, the returned program p^* will satisfy the following condition:

$$\min_{\boldsymbol{x} \in X^n} U(\mathcal{L}_o(p^*[\boldsymbol{x}], \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{x}, \boldsymbol{x}_c), C(p^*)) - \min_{p \in G, \boldsymbol{x} \in X^n} U(\mathcal{L}_o(p[\boldsymbol{x}], \boldsymbol{y}_c), \mathcal{L}_i(\boldsymbol{x}, \boldsymbol{x}_c), C(p)) \leq \epsilon$$

□

8.3 Implementation

I have implemented my modified algorithm within the *Rose* synthesis tool. *Rose* is parameterized over a large class of objective functions, loss functions, and complexity measures. I benchmark *Rose* using the SyGuS 2018 benchmark suite [3].

Domain Specific Language and Abstractions: I use a modified version of the string processing domain specific language from [42, 22] (Figure 8-6), which supports extracting substrings (using the SubStr function) of the input string x.The function SubStr function extracts a substring using a start and an end position. A position is defined using a constant index (ConstPos).

$$
\begin{array}{rcl}
\text{String expr } e & := & \mathsf{Str}(f); \\
\text{Substring expr } f & := & \mathsf{ConstStr}(s) \mid \mathsf{SubStr}(x, p_1, p_2); \\
\text{Position } p & := & \mathsf{ConstPos}(k);
\end{array}
$$

Figure 8-6: DSL for string transformations where k is an integer and s is a string constant.

$$
\begin{array}{rcl}
[\![f(s_1 = c_1, \ldots, s_k = c_k)]\!]^{\#} & := & (s = [\![f(c_1, \ldots c_k)]\!]) \\
[\![\mathsf{Str}(p)]\!]^{\#} & := & p \\
[\![\mathsf{SubStr}(p, p_1 = i_2, p_3 = i_3)]\!]^{\#} & := & (len(e) = i_3 - i_1 + 1) \\
[\![\mathsf{SubStr}(e[i_1] = c, p_1 = i_2, p_3 = i_3)]\!]^{\#} & := & (e[i_1 - i_2] = c) \text{ if } i_2 \leq i_1 \leq i_3
\end{array}
$$

Figure 8-7: Abstract semantics for string transformation DSL.

Abstractions: I use the universe of predicates, initial abstractions, and abstract loss function semantics presented in Chapter 6 (Section 6.7). I present an expanded version of abstract semantics for my DSL in Figure 8-7.

8.4 Experimental Results

I use the SyGuS 2018 benchmark suite [1] to benchmark my technique. My current implementation doesn't support pattern matching. I use the size complexity measure $\mathsf{Size}(p)$ (Subsection 2.2.4) and uniform regularizer $\mathcal{R}_U$ (Definition 4) for these experiments.

8.4.1 Scalability

I evaluate the scalability of my implementation by applying it to all problems in the SyGuS 2018 benchmark suite [1], which do not require pattern matching. There are a total of 12 such problems within the SyGuS benchmark suite.

For each problem, I use the clean (noise-free) dataset for the problem provided with the benchmark suite. All experiments are run on an 3.00 GHz Intel(R) Xeon(R) CPU E5-2690 v2 machine with 512GB memory running Linux 4.15.0. I use a timeout limit of 10 minutes and bounded scope height threshold of 4.

I evaluated my implementation for these 12 benchmark problems on five different loss functions, $0/\infty$ loss function $\mathcal{L}_{0/\infty}$, 0/1 loss function $\mathcal{L}_{0/1}$, Damerau-Levenstein loss function $\mathcal{L}_{DL}$, 1-Delete loss function $\mathcal{L}_{1D}$, and n-Substitution loss function $\mathcal{L}_{nS}$.

These benchmark problem contain 6-400 input-output examples. For each experiment, I selected a benchmark problem, an input loss function out of these five loss functions, and a output loss function out of these five loss functions. For each combination of the benchmark problem, input loss function, and output loss function, my implementation synthesizes the correct program in less than 0.1 second.

8.4.2 Noisy Datasets, Character Replacements

I next present results for my implementation running on data sets with character replacements. I use the phone, phone-1, phone_short, and phone-1_short benchmarks problems for this experiment. These benchmark problems contain 6 input-output examples each.

For each benchmark, I introduce both input and output noise. I use a noise source which cyclically substitutes a single character for each input and output in turn. For output strings, the noise source starts by corrupting the first character for the output string in the first input-output example, incrementing the position it corrupts by 1 for the next input-output example, and then wrapping around to the first position again. The noise source follows a same process for input strings but starts by corrupting the second character for the first input string. To construct a noisy dataset with k uncorrupted input-output examples, I first corrupt the entire dataset and then I replace the first k corrupted input-output examples with the original noise-free input-output examples. For these experiments I use the same loss function as both the input loss function and output loss function. I run this experiment for two loss functions, n-Substitution loss function and Damerau-Levenshtein loss function.

Figure 8-8 presents the results of this experiment. The first column represents the number of uncorrupted input-output in the noisy dataset. The next columns present the time it takes for my synthesis algorithm to terminate (in milliseconds) for four benchmark problems. For each benchmark problem, I present the runtime for two different loss functions, the n-Substitution loss function (nS) and the Damerau-Levenshtein loss function (DL).

For each combination, my technique is able to synthesize the correct program even when all input-output examples were corrupted. Our results showcase that decreasing

Number of uncorrupted Examples	phone		phone_short		phone-1		phone-1_short	
	nS	DL	nS	DL	nS	DL	nS	DL
0	4365	52692	3269	47812	6161	76464	5859	75247
1	3168	16895	2078	13978	2733	47689	2800	49182
2	741	1626	621	1683	3287	34600	2576	33849
3	455	1888	235	460	797	5906	598	6314
4	216	808	271	244	319	900	167	800
5	55	112	21	171	131	242	68	119
6	39	25	21	16	3	4	2	3

Figure 8-8: Runtime for phone, phone_small, phone-1, and phone-1_small benchmark problems for character replacement based input and output noise.

the amount of noise in the dataset improves the performance of the synthesis algorithm for both, the n-Substitution loss function and the Damerau-Levenshtein loss function. The synthesis using the n-Substitution loss function is also, on an average, 6.5 times faster compared to Damerau-Levenshtein loss function.

Chapter 9

Related Work

Program synthesis has received much attention from programming languages community [26, 4, 13, 14, 29, 36, 48, 7, 2, 37, 39, 38, 41, 40]. I can identity two research directions within the program synthesis literature that are related to this work. I also discuss work within learning theory related to my work.

9.1 Programming-by-Example

I first discuss literature related to noise-free program synthesis over datasets. The major difference between my work and these research directions is that these systems require all input-output examples to be correct/noise-free.

Synthesis Systems Using Solvers: These systems use a solver (for example, a SAT solver) to synthesize programs, given input output examples [24]. To the best of my knowledge, there does not exist any solver based synthesis system, which can synthesize programs over noisy data.

Enumerative Techniques: These techniques search the space of programs to find a single program that is consistent with the given examples [16, 27]. Specifically, they enumerate all programs in the given DSL and terminate when they find the correct program. These techniques may apply different heuristics/techniques to prune the search space/speed up this process [27].

VSA-based/Tree Automata-based Techniques: These techniques build complex data structures representing all possible programs compatible with the given

examples [35, 30, 43]. My work modifies these techniques to handle noisy data and to synthesize programs that minimize an objective function over noisy dataset.

Abstraction-Refinement based Synthesis Algorithms: There has been work done on using abstraction refinement/refinement types to synthesize programs [43]. Given a noise-free dataset and a program, checking if a program is correct or incorrect simply checks if the synthesized programs satisfy all input/output examples. To refine an abstraction, these techniques construct a proof of incorrectness. Each abstraction identifies a set of programs, some of which may be correct and others of which may be incorrect. Refinement first identifies a program that does not satisfy one or more of the input/output examples, then generates constraints that refine the abstraction to eliminate this program. Iterative refinement eventually produces the final program.

Abstraction in my noisy program synthesis framework, in contrast, works with an abstraction that approximates the loss function over a set of programs. The refinement step selects a program within the abstraction space, computes its loss, then uses this computed loss to refine the loss approximation to bring this approximation closer to the actual loss. This refinement step, in expectation, reduces the inaccuracy in the approximated loss function of the programs identified by the abstraction. In contrast to previous approaches, which work with abstractions based on program correctness and refinement steps that eliminate incorrect programs, my approach works with abstractions that maintain a sound, conservative approximation of the minimum loss function over the set of programs identified by the abstraction and refinement steps that eliminate programs based on the loss of the programs.

One key difference is that refinement steps in previous techniques rely on the ability to identify correct and incorrect programs. Because our technique works with noisy datasets, it can never tell if a candidate program has minimal loss without comparing the program to all other current candidate programs (unless the loss happens to be zero). It instead uses abstract minimum loss values to bound how far off the optimal loss any candidate program may be. Instead of working with correct or incorrect programs, my technique works by iteratively improving the accuracy of the minimum loss function estimation captured by the abstraction.

My technique therefore combines abstract tree automata with an abstraction-based optimization process. My approach, in contrast to previous approaches that use abstract tree automata, enables us to synthesize programs that optimize an objective function over a set of noisy input/output examples, including synthesizing correct programs that may disagree with one, some, or even all of the provided input/output examples.

[42] uses abstract tree automata and abstraction refinement for program synthesis. Because their refinement strategy prunes any program that does not satisfy all of the provided input/output examples, their algorithm requires the dataset to be noise-free. This pruning is necessary as this allows their technique to effectively capture constraints to prune large part of the search space.

Neural Program Synthesis/Machine-Learning Approaches: Researchers have investigated techniques that use machine learning/deep neural networks to synthesize programs [32, 11, 5]. The techniques primarily focus on synthesizing programs over noise-free datasets. These techniques require a training phase and a differentable loss function and provide no guarantees that the synthesized program will minimize the objective function. My technique, in contrast, does not require a training phase, can work with arbitrary loss functions including, for example, the Damerau-Levenshtien loss function, and comes with a guarantee that the synthesized program will minimize the objective function over the provided (noisy) input/output examples.

9.2 Techniques to Tolerate Data Corruptions

I next discuss program synthesis techniques that deal with data corruptions.

Data Set Sampling or Cleaning: There has been recent work which aspires to clean the dataset or pick representative examples from the dataset for synthesis [20, 32, 31], for example by using machine learning or data cleaning to select productive subsets of the dataset over which to perform exact synthesis. In contrast to these techniques, my proposed techniques 1) provide deterministic guarantees (as opposed to either probabilistic guarantees as in [32] or no guarantees at all as in [31, 20]), 2) do not require the use of oracles as in [32], 3) can operate successfully even on

datasets in which most or even all of the input-output examples are corrupted, and 4) do not require the explicit selection of a subset of the dataset to drive the synthesis as in [20, 32].

Best-Effort Program Synthesis: In a work done concurrently with ours [28], Peleg et al. present an enumeration based technique to synthesis programs from input-output datasets containing some incorrect outputs. Their technique returns a ranked list of partially valid programs, removing programs which are observationally equivalent. Their technique uses a fixed fitness function to order these partial results. My framework subsumes theirs. By using the following loss function $\mathcal{L}$, complexity metric C, and objective function $U(l, c) = l + c$, program synthesized by my technique will be the top result of their ranked list.

$$\mathcal{L}(p, \mathcal{D}) = 3 \times \mathcal{L}_{0/1}(p, \mathcal{D}) + \mathcal{L}_{DL}(p, \mathcal{D})$$

$$C(p) = 2 \times \mathsf{relevancy}(p) + \mathsf{size}(p)$$

Given some prior knowledge about the noise process, my techniques allows the user to capture information even from corrupted input-output examples, specifically by developing appropriate loss functions, complexity measures, and objective functions. As showcased within my results section, a suitable loss function allows us to synthesize the correct program even when all input-output examples are corrupted.

Since Peleg et al.'s work prioritizes programs which fit the largest set of input-output examples, their technique will always return the wrong solution when all input-output examples are corrupted. Even when all input-output examples are not corrupted, their technique may synthesize a program which satisfies a large set of corrupted input-output examples. The program which satisfies these corrupted input-output examples may not be the correct hidden program (i.e., the program which generated the original input-output examples, which was corrupted). My technique on the other hand, given an appropriate loss function, can return the correct program, even though it may satisfy a smaller set of input-output examples.

Peleg el al.'s technique provides no capability to tradeoff between accuracy and

complexity of the synthesized program. Using the tradeoff framework, our technique also allows a user to tradeoff between a synthesizing a simpler solution by sacrificing accuracy or synthesize a complex program which overfits the dataset. The tradeoff framework is frequently used in machine learning to prevent overfitting of learned models.

Bayesian Program Synthesis: There has been previous attempts to apply Bayesian inference to synthesize programs in presence of noise [12] or synthesize probabilistic programs [33, 44]. These papers present different techniques to synthesize programs, when the noise source is known apriori. They do not discuss the correctness of their technique in the absence of perfect information about the noise source, nor do they discuss the conditions under which their algorithms will converge to the correct hidden program with high probability.

My research uses Bayesian concepts to connect noisy program synthesis algorithms, which use loss functions and complexity measures, to noise source, the input source, and the program source. In contrast to these previous papers, I discuss conditions under which the synthesis algorithm will converge to the correct program with high probability. I also formalize the optimal parameters for these loss function based noisy program synthesis algorithms using concepts from Bayesian Inference.

9.3 Connections to Learning Theory

Learning theory captures the formal aspects of learning models over noisy data [23, 6, 25]. My work takes concepts from learning theory and applies them to the specific context of synthesizing programs over noisy data. To the best of my knowledge, the special case of noisy program synthesis has never been explored in learning theory.

There has considerable work done on designing loss functions for training neural networks [46, 18]. To the best of my knowledge, no such work exists on designing loss functions for noisy program synthesis.

Chapter 10

Conclusion

Dealing with noisy data is a pervasive problem in modern computing environments. Previous program synthesis systems target datasets in which all input-output examples are correct to synthesize programs that match all input-output examples in the dataset.

I formalize the problem of program synthesis over datasets containing noise as an optimization problem. I present synthesis algorithms which successfully synthesize programs over noisy data. These algorithms synthesize programs in the presence of output noise, input noise, and even over domain specific languages containing an infinite set of constants.

The results highlight how these techniques, by exploiting information from a variety of sources — structure from the underlying DSL, information left intact by noise sources — can deliver effective program synthesis even in the presence of substantial noise.

I also formally define the concepts of an optimal loss function and convergence. Building on these concepts, I characterize a large range of potential noise sources, corresponding optimal loss functions, and the conditions under which we can expect noisy program synthesis algorithms to converge. These results provide insight into the empirical results.

Future Directions

I will now discuss some potential future research directions.

Application Domains: In this thesis I work with a DSL containing string transforming programs. One future direction is extending this framework to other domains containing noisy data. Potential examples include the domains of spreadsheets [43] and matrix transformations [42]. These domains contain examples specified by a human user. These examples are amenable to human error. Our framework can be used to make these techniques robust to human errors.

New Techniques: There are concepts used by other noise-free synthesis algorithms which may be exploited to construct new noisy program synthesis algorithms. Neural Network based techniques [13] may be a good starting point as noisy program synthesis algorithms can use neural networks to improve their search process.

Motivated by machine learning, a stochastic version of the abstraction refinement based optimization, which works with subsets of the input-output examples, may speed up the synthesis process. Stochastic abstraction refinement based optimization may even reduce the size of the abstract finite tree automaton constructed by our technique.

Theoretical Questions: Optimal loss functions and convergence for program synthesis with noisy inputs is left unanswered and is a potential direction for future work.

Bibliography

[1] Sygus 2018 string benchmark suite. `https://github.com/SyGuS-Org/benchmarks/tree/master/comp/2019/PBE_SLIA_Track/from_2018`, 2018. Accessed: 2020-07-18.

[2] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8. IEEE, 2013.

[3] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8. IEEE, 2013.

[4] Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. Search-based program synthesis. *Commun. ACM*, 61(12):84–93, 2018.

[5] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

[6] Robert C Bolles. Learning theory. 1975.

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

[8] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications, 2008.

[9] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[10] Fred J. Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176, March 1964.

[11] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdelrahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 990–998. JMLR. org, 2017.

[12] Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. Unsupervised learning by program synthesis. 2015.

[13] Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. Sampling for bayesian program learning. In Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 1289–1297, 2016.

[14] Kevin Ellis, Armando Solar-Lezama, and Joshua B. Tenenbaum. Unsupervised learning by program synthesis. In Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 973–981, 2015.

[15] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In *ACM SIGPLAN Notices*, volume 52, pages 422–436. ACM, 2017.

[16] John K Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *ACM SIGPLAN Notices*, volume 50, pages 229–239. ACM, 2015.

[17] Andrew Gelman, John B Carlin, Hal S Stern, David B Dunson, Aki Vehtari, and Donald B Rubin. *Bayesian data analysis*. CRC press, 2013.

[18] Aritra Ghosh, Himanshu Kumar, and PS Sastry. Robust loss functions under label noise for deep neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[19] Peter Grassberger, Rainer Hegger, Holger Kantz, Carsten Schaffrath, and Thomas Schreiber. On noise reduction methods for chaotic data. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 3(2):127–141, 1993.

[20] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM Sigplan Notices*, volume 46, pages 317–330. ACM, 2011.

[21] Shivani Gupta and Atul Gupta. Dealing with noise problem in machine learning data-sets: A systematic review. *Procedia Computer Science*, 161:466–474, 2019.

[22] Shivam Handa and Martin C Rinard. Inductive program synthesis over noisy data. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 87–98, 2020.

[23] Knud Illeris. An overview of the history of learning theory. *European Journal of Education*, 53(1):86–101, 2018.

[24] Shachar Itzhaky, Sumit Gulwani, Neil Immerman, and Mooly Sagiv. A simple inductive synthesis methodology and its applications. In *ACM Sigplan Notices*, volume 45, pages 36–46. ACM, 2010.

[25] Michael J Kearns, Umesh Virkumar Vazirani, and Umesh Vazirani. *An introduction to computational learning theory*. MIT press, 1994.

[26] Madhav Khirwar. Wake-sleep bayesian program synthesis applications in bioinformatics. 2021.

[27] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. *ACM SIGPLAN Notices*, 50(6):619–630, 2015.

[28] Hila Peleg and Nadia Polikarpova. Perfect is the enemy of good: Best-effort program synthesis. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[29] Nadia Polikarpova and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. *CoRR*, abs/1510.08419, 2015.

[30] Oleksandr Polozov and Sumit Gulwani. Flashmeta: a framework for inductive program synthesis. In *ACM SIGPLAN Notices*, volume 50, pages 107–126. ACM, 2015.

[31] Yewen Pu, Zachery Miranda, Armando Solar-Lezama, and Leslie Pack Kaelbling. Selecting representative examples for program synthesis. *arXiv preprint arXiv:1711.03243*, 2017.

[32] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. Learning programs from noisy data. In *ACM SIGPLAN Notices*, volume 51, pages 761–774. ACM, 2016.

[33] Feras A Saad, Marco F Cusumano-Towner, Ulrich Schaechtle, Martin C Rinard, and Vikash K Mansinghka. Bayesian synthesis of probabilistic programs for automatic data modeling. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–32, 2019.

[34] Thomas Schreiber and Peter Grassberger. A simple noise-reduction method for real data. *Physics letters A*, 160(5):411–418, 1991.

[35] Rishabh Singh and Sumit Gulwani. Transforming spreadsheet data types using examples. In *Acm Sigplan Notices*, volume 51, pages 343–356. ACM, 2016.

[36] Armando Solar-Lezama. The sketching approach to program synthesis. In Zhenjiang Hu, editor, *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*, volume 5904 of *Lecture Notes in Computer Science*, pages 4–13. Springer, 2009.

[37] Armando Solar-Lezama. Program sketching. *Int. J. Softw. Tools Technol. Transf.*, 15(5-6):475–495, 2013.

[38] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodík, Vijay A. Saraswat, and Sanjit A. Seshia. Sketching stencils. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 167–178. ACM, 2007.

[39] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. Sketching concurrent data structures. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 136–148. ACM, 2008.

[40] Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 281–294. ACM, 2005.

[41] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In John Paul Shen and Margaret Martonosi, editors, *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 404–415. ACM, 2006.

[42] Xinyu Wang, Isil Dillig, and Rishabh Singh. Program synthesis using abstraction refinement. *Proceedings of the ACM on Programming Languages*, 2(POPL):63, 2017.

[43] Xinyu Wang, Isil Dillig, and Rishabh Singh. Synthesis of data completion scripts using finite tree automata. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):62, 2017.

[44] Sam Witty, Alexander Lew, David Jensen, and Vikash Mansinghka. Bayesian causal inference via probabilistic program synthesis. *arXiv preprint arXiv:1910.14124*, 2019.

[45] Hui Xiong, Gaurav Pandey, Michael Steinbach, and Vipin Kumar. Enhancing data analysis with noise removal. *IEEE Transactions on Knowledge and Data Engineering*, 18(3):304–319, 2006.

[46] Yilun Xu, Peng Cao, Yuqing Kong, and Yizhou Wang. L_dmi: An information-theoretic noise-robust loss function. *arXiv preprint arXiv:1909.03388*, 2019.

[47] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. Synthesizing transformations on hierarchically structured data. In *ACM SIGPLAN Notices*, volume 51, pages 508–521. ACM, 2016.

[48] Kuat Yessenov, Zhilei Xu, and Armando Solar-Lezama. Data-driven synthesis for object-oriented frameworks. In Cristina Videira Lopes and Kathleen Fisher, editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 65–82. ACM, 2011.