Inductive Program Synthesis over Noisy Data

Shivam Handa
Electrical Engineering and Computer Science
Massachusetts Institute of Technology
U.S.A.
shivam@mit.edu

Martin C. Rinard
Electrical Engineering and Computer Science
Massachusetts Institute of Technology
U.S.A.
rinard@csail.mit.edu

ABSTRACT

We 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. This framework is based on an extension of finite tree automata called *state-weighted finite tree automata*. We show how to apply this framework to formulate and solve a variety of program synthesis problems over noisy data. Results from our implemented system running on problems from the SyGuS 2018 benchmark suite highlight its ability to successfully synthesize programs in the face of noisy data sets, including the ability to synthesize a correct program even when every input-output example in the data set is corrupted.

CCS CONCEPTS

• **Theory of computation** → *Formal languages and automata theory*; • **Software and its engineering** → **Programming by example**; • **Computing methodologies** → **Machine learning**.

KEYWORDS

Program Synthesis, Noisy Data, Corrupted Data, Machine Learning

ACM Reference Format:
Shivam Handa and Martin C. Rinard. 2020. Inductive Program Synthesis over Noisy Data. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3409732

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 [10, 18], data wrangling [7], data completion [20], and data structure manipulation [8, 12, 21]. Even though these efforts have been largely successful, they do not aspire to work with noisy data sets that may contain corrupted input-output examples.

We present a new program synthesis technique that is designed to work with noisy/corrupted data sets. Given:

- **Programs:** A collection of programs p defined by a grammar G and a bounded scope threshold d,
- **Data Set:** A data set $\mathcal{D} = \{(\sigma_1, o_1), \ldots, (\sigma_n, o_n)\}$ of input-output examples,
- **Loss Function:** A loss function $\mathcal{L}(p, \mathcal{D})$ that measures the cost of the input-output examples on which p produces a different output than the output in the data set $\mathcal{D}$,
- **Complexity Measure:** A complexity measure $C(p)$ that measures the complexity of a given program p,
- **Objective Function:** An arbitrary objective function $U(l, c)$, which maps loss l and complexity c to a totally ordered set, such that for all values of l, $U(l, c)$ is monotonically nondecreasing with respect to c,

our program synthesis technique produces a program p that minimizes $U(\mathcal{L}(p, \mathcal{D}), C(p))$. Example problems that our program synthesis technique can solve include:

- **Best Fit Program Synthesis:** Given a potentially noisy data set $\mathcal{D}$, find a *best fit* program p for $\mathcal{D}$, i.e., a p that minimizes $\mathcal{L}(p, \mathcal{D})$.
- **Accuracy vs. Complexity Tradeoff:** Given a data set $\mathcal{D}$, find p that minimizes the weighted sum $\mathcal{L}(p, \mathcal{D}) + \lambda \cdot C(p)$. This problem enables the programmer to define and minimize a weighted tradeoff between the complexity of the program and the loss.
- **Data Cleaning and Correction:** Given a data set $\mathcal{D}$, find p that minimizes the loss $\mathcal{L}(p, \mathcal{D})$. Input-output examples with nonzero loss are identified as corrupted and either 1) filtered out or 2) replaced with the output from the synthesized program.
- **Bayesian Program Synthesis:** Given a data set $\mathcal{D}$ and a probability distribution $\pi(p)$ over programs p, find the most probable program p given $\mathcal{D}$.
- **Best Program for Given Accuracy:** Given a data set $\mathcal{D}$ and a bound b, find p that minimizes $C(p)$ subject to $\mathcal{L}(p, \mathcal{D}) \leq b$. One use case finds the simplest program that agrees with the data set $\mathcal{D}$ on at least $n - b$ input-output examples.
- **Forced Accuracy:** Given data sets $\mathcal{D}'$, $\mathcal{D}$, where $\mathcal{D}' \subseteq \mathcal{D}$, find p that minimizes the weighted sum $\mathcal{L}(p, \mathcal{D}) + \lambda \cdot C(p)$ subject to $\mathcal{L}(p, \mathcal{D}') \leq b$. One use case finds a program p which minimizes the loss over the data set $\mathcal{D}$ but is always correct for $\mathcal{D}'$.
- **Approximate Program Synthesis:** Given a clean (noise-free) data set $\mathcal{D}$, find the least complex program p that minimizes the loss $\mathcal{L}(p, \mathcal{D})$. Here the goal is not to work with a noisy data set, but instead to find the best approximate solution to a synthesis problem when an exact solution does not exist within the collection of considered programs p.

1.1 Noise Models and Discrete Noise Sources

We work with noise models that assume a (hidden) clean data set combined with a noise source that delivers the noisy data set presented to the program synthesis system. Like many inductive program synthesis systems [10, 18], one target is *discrete problems*

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11.
https://doi.org/10.1145/3368089.3409732

that involve discrete data such as strings, data structures, or tablular data. In contrast to traditional machine learning problems, which often involve continuous noise sources [4], the noise sources for discrete problems often involve *discrete noise* — noise that involves a discrete change that affects only part of each output, leaving the remaining parts intact and uncorrupted.

1.2 Loss Functions and Use Cases

Different loss functions can be appropriate for different noise sources and use cases. The 0/1 loss function, which counts the number of input-output examples where the data set $\mathcal{D}$ and synthesized program p do not agree, is a general loss function that can be appropriate when the focus is to maximize the number of inputs for which the synthesized program p produces the correct output. The Damerau-Levenshtein (DL) loss function [5], which measures the edit difference under character insertions, deletions, substitutions, and/or transpositions, extracts information present in partially corrupted outputs and can be appropriate for measuring discrete noise in input-output examples involving text strings. The $0/\infty$ loss function, which is ∞ unless p agrees with all of the input-output examples in the data set $\mathcal{D}$, specializes our technique to the standard program synthesis scenario that requires the synthesized program to agree with all input-output examples.

Because discrete noise sources often leave parts of corrupted outputs intact, *exact program synthesis* (i.e., synthesizing a program that agrees with all outputs in the hidden clean data set) is often possible even when all outputs in the data set are corrupted. Our experimental results (Section 9) indicate that matching the loss function to the characteristics of the discrete noise source can enable very accurate program synthesis even when 1) there are only a handful of input-output examples in the data and 2) all of the outputs in the data set are corrupted. We attribute this success to the ability of our synthesis technique, working in conjunction with an appropriately designed loss function, to effectively extract information from outputs corrupted by discrete noise sources.

1.3 Technique

Our technique augments finite tree automata (FTA) to associate accepting states with weights that capture the loss for the output associated with each accepting state. Given a data set $\mathcal{D}$, the resulting *state-weighted finite tree automata* (SFTA) partition the programs p defined by the grammar G into equivalence classes. Each equivalence class consists of all programs with identical input-output behavior on the inputs in $\mathcal{D}$. All programs in a given equivalence class therefore have the same loss over $\mathcal{D}$. The technique then uses dynamic programming to find the minimum complexity program p in each equivalence class [9]. From this set of minimum complexity programs, the technique then finds the program p that minimizes the objective function $U(p, \mathcal{D})$.

1.4 Experimental Results

We have implemented our technique and applied it to various programs in the SyGuS 2018 benchmark set [1]. 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 (in this paper we consider a synthesized program to be correct if it agrees with all input-output examples in the original hidden clean data set). Even with the 0/1 loss function, which does not aspire to extract any information from corrupted outputs, the technique is typically able to synthesize a correct program even with only a few correct (uncorrupted) input-output examples in the data set. Overall the results highlight the potential for effective program synthesis even in the presence of substantial noise.

1.5 Contributions

This paper makes the following contributions:

- **Technique:** It presents an implemented technique for inductive program synthesis over noisy data. The technique uses an extension of finite tree automata, *state-weighted finite tree automata*, to synthesize programs that minimize an objective function involving the loss over the input data set and the complexity of the synthesized program.
- **Use Cases:** It presents multiple uses cases including best fit program synthesis for noisy data sets, navigating accuracy vs. complexity tradeoffs, Bayesian program synthesis, identifying and correcting corrupted data, and approximate program synthesis.
- **Experimental Results:** It presents experimental results from our implemented system 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, 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.

2 PRELIMINARIES

We next review finite tree automata (FTA) and FTA-based inductive program synthesis.

2.1 Finite Tree Automata

Finite tree automata are a type of state machine which accept trees rather than strings. They generalize standard finite automata to describe a regular language over trees.

Definition 1 (**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* $f(q_1, \ldots, q_k) \rightarrow q$ *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 (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*

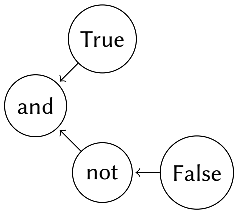

Figure 1: Tree for formula and(True, not(False))

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

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

Figure 2: Execution semantics for program p

$Q_f = \{q_T\}$ *and the following transition rules* Δ*:*

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

The above tree automaton accepts all propositional logic formulas over True and False which evaluate to True. Figure 1 presents the tree for the formula and(True, not(False)).

2.2 Domain Specific Languages (DSLs)

We next define the programs we consider, how inputs to the program are specified, and the program semantics. Without loss of generality, we assume programs p are specified as parse trees in a domain-specific language (DSL) grammar G. Internal nodes represent function invocations; leaves are constants/0-arity symbols in the DSL. A program p executes in an input σ. $[\![p]\!]\sigma$ denotes the output of p on input σ ($[\![.]\!]$ is defined in Figure 2).

All valid programs (which can be executed) are defined by a DSL grammar $G = (T, N, P, s_0)$ where:

- T is a set of terminal symbols. These may include constants and symbols which may change value depending on the input σ.
- 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.

We 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 paper can be generalized to any DSL which can be specified within the above framework.

Example 2. *The following DSL defines expressions over input x, constants 2 and 3, and addition and multiplication:*

$$\begin{array}{rcl}
n & := & x \mid n + t \mid n \times t; \\
t & := & 2 \mid 3;
\end{array}$$

$$\frac{t \in T,\ [\![t]\!]\sigma = c}{q_t^c \in Q}\ (\textsc{Term}) \qquad \frac{q_{s_0}^o \in Q}{q_{s_0}^o \in Q_f}\ (\textsc{Final})$$

$$\frac{\begin{array}{c} s \to f(s_1, \ldots s_k) \in P,\ \{q_{s_1}^{c_1}, \ldots, q_{s_k}^{c_k}\} \subseteq Q, \\ [\![f(c_1, \ldots c_k)]\!]\sigma = c \end{array}}{q_s^c \in Q,\ f(q_{s_1}^{c_1}, \ldots, q_{s_k}^{c_k}) \to q_s^c \in \Delta}\ (\textsc{Prod})$$

Figure 3: Rules for constructing a CFTA $\mathcal{A} = (Q, F, Q_f, \Delta)$ **given input** σ**, output** o**, and grammar** $G = (T, N, P, s_0)$**.**

2.3 Concrete Finite Tree Automata

We review the approach introduced by [19, 20] to use 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 an input-output example (σ, o) and DSL $(G, [\![.]\!])$, construct a CFTA using the rules in Figure 3. The alphabet of the CFTA contains built-in functions within the DSL. The states in the CFTA are of the form q_s^c, where s is a symbol (terminal or non-terminal) in G and c is a concrete value. The existence of state q_s^c implies that there exists a partial program which can map σ to concrete value c. Similarly, the existence of transition $f(q_{s_1}^{c_1}, q_{s_2}^{c_2} \ldots q_{s_k}^{c_k}) \to q_s^c$ implies $f(c_1, c_2 \ldots c_k) = c$.

The Term rule states that if we have a terminal t (either a constant in our language or input symbol x), execute it with the input σ and construct a state q_t^c (where $c = [\![t]\!]\sigma$). The Final rule states that, given start symbol s_0 and we expect o as the output, if $q_{s_0}^o$ exists, then we have an accepting state. The Prod rule states that, if we have a production rule $f(s_1, s_2, \ldots s_k) \to s \in \Delta$, and there exists states $q_{s_1}^{c_1}, q_{s_2}^{c_2} \ldots q_{s_k}^{c_k} \in Q$, then we also have state q_s^c in the CFTA and a transition $f(q_{s_1}^{c_1}, q_{s_2}^{c_2}, \ldots q_{s_k}^{c_k}) \to q_s^c$.

The language of the CFTA constructed from Figure 3 is exactly the set of parse trees of DSL programs that are consistent with our input-output example (i.e., maps input σ to output o).

In general, the rules in Figure 3 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 it will accept is larger than a given threshold d. This results in a CFTA which accepts all programs which are consistent with the input-output example but are smaller than the given threshold (it may accept some programs which are larger than the given threshold but it will never accept a program which is inconsistent with the input-output example). This is standard practice in the synthesis literature [13, 19].

2.4 Intersection of Two CFTAs

Given two CFTAs $\mathcal{A}_1$ and $\mathcal{A}_2$ built over the same grammar G from input-output examples (σ_1, o_1) and (σ_2, o_2) respectively, the intersection of these two automata $\mathcal{A}$ contains programs which satisfy both input-output examples (or has the empty language). Given CFTAs $\mathcal{A} = (Q, F, Q_f, \Delta)$ and $\mathcal{A}' = (Q', F', Q_f', \Delta)$, $\mathcal{A}^* =$

$(Q^*, F, Q_f^*, \Delta^*)$ is the intersection of $\mathcal{A}$ and $\mathcal{A}'$, where Q^*, Q_f^*, and Δ^* are the smallest set such that:

$$q_s^{\vec{c_1}} \in Q \text{ and } q_s^{\vec{c_2}} \in Q' \text{ then } q_s^{\vec{c_1}:\vec{c_2}} \in Q^*$$

$$q_s^{\vec{c_1}} \in Q_f \text{ and } q_s^{\vec{c_2}} \in Q_f' \text{ then } q_s^{\vec{c_1}:\vec{c_2}} \in Q_f^*$$

$$f(q_{s_1}^{\vec{c_1}}, \dots q_{s_k}^{\vec{c_k}}) \rightarrow q_s^{\vec{c}} \in \Delta \text{ and } f(q_{s_1}^{\vec{c_1'}}, \dots q_{s_k}^{\vec{c_k'}}) \rightarrow q_s^{\vec{c'}} \in \Delta'$$

$$\text{then } f(q_{s_1}^{\vec{c_1}:\vec{c_1'}}, \dots q_{s_k}^{\vec{c_k}:\vec{c_k'}}) \rightarrow q_s^{\vec{c}:\vec{c'}} \in \Delta^*$$

where $\vec{c}$ denotes a vector of values and $\vec{c_1} : \vec{c_2}$ denote a vector constructed by appending vector $\vec{v_2}$ at the end of vector $\vec{v_1}$.

3 LOSS FUNCTIONS

Given a data set $\mathcal{D} = \{(\sigma_1, o_1), \dots, (\sigma_n, o_n)\}$ and a program p, a **Loss Function** $\mathcal{L}(p, \mathcal{D})$ calculates how incorrect the program is with respect to the given data set. We work with loss functions $\mathcal{L}(p, \mathcal{D})$ that depend only on the data set and the outputs of the program for the inputs in the data set, i.e., given programs p_1, p_2, such that for all $(\sigma_i, o_i) \in \mathcal{D}$, $[\![p_1]\!]\sigma_i = [\![p_2]\!]\sigma_i$, then $\mathcal{L}(p_1, \mathcal{D}) = \mathcal{L}(p_2, \mathcal{D})$. We also further assume that the loss function $\mathcal{L}(p, \mathcal{D})$ can be expressed in the following form:

$$\mathcal{L}(p, \mathcal{D}) = \sum_{i=1}^{n} L(o_i, [\![p]\!]\sigma_i)$$

where $L(o_i, [\![p]\!]\sigma_i)$ is a per-example loss function.

Definition 2. 0/1 **Loss Function:** *The* 0/1 *loss function* $\mathcal{L}_{0/1}(p, \mathcal{D})$ *counts the number of input-output examples where p does not agree with the data set* $\mathcal{D}$*:*

$$\mathcal{L}_{0/1}(p, \mathcal{D}) = \sum_{i=1}^{n} 1 \text{ if } (o_i \neq [\![p]\!]\sigma_i) \text{ else } 0$$

Definition 3. $0/\infty$ **Loss Function:** *The* $0/\infty$ *loss function* $\mathcal{L}_{0/\infty}(p, \mathcal{D})$ *is 0 if p matches all outputs in the data set* $\mathcal{D}$ *and* ∞ *otherwise:*

$$\mathcal{L}_{0/\infty}(p, \mathcal{D}) = 0 \text{ if } (\forall(\sigma, o) \in \mathcal{D}.o = [\![p]\!]\sigma) \text{ else } \infty$$

Definition 4. **Damerau-Levenshtein (DL) Loss Function:** *The DL loss function* $\mathcal{L}_{DL}(p, \mathcal{D})$ *uses the Damerau-Levenshtein metric [5], to measure the distance between the output from the synthesized program and the corresponding output in the noisy data set:*

$$\mathcal{L}_{DL}(p, \mathcal{D}) = \sum_{(\sigma_i, o_i) \in \mathcal{D}} L_{[\![p]\!]\sigma_i, o_i}\Big(|[\![p]\!]\sigma_i|, |o_i|\Big)$$

where, $L_{a,b}(i, j)$ *is the Damerau-Levenshtein metric [5].*

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 [5], the DL loss function may be appropriate for computations that work with human-provided text input-output examples.

4 COMPLEXITY MEASURE

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

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

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

Given an FTA $\mathcal{A}$, we can use dynamic programming to find the minimum complexity parse tree (under the above $\text{Cost}(p)$ measure) accepted by $\mathcal{A}$ [9]. In general, given an FTA $\mathcal{A}$, we assume we are provided with a method to find the program p accepted by $\mathcal{A}$ which minimizes the complexity measure.

5 OBJECTIVE FUNCTIONS

Given loss l and complexity c, an **Objective Function** $U(l, c)$ maps l, c to a totally ordered set such that for all l, $U(l, c)$ is monotonically nondecreasing with respect to c.

Definition 5. **Tradeoff Objective Function:** *Given a tradeoff parameter* $\lambda > 0$*, the tradeoff objective function* $U_\lambda(l, c) = l + \lambda c$*.*

This objective function trades the loss of the synthesized program off against 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 6. **Lexicographic Objective Function:** *A lexicographic objective function* $U_L(l, c) = \langle l, c \rangle$ *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 loss, then the complexity. It may be appropriate, for example, for best fit program synthesis, data cleaning and correction, and approximate program synthesis over clean data sets.

6 STATE-WEIGHTED FTA

State-weighted finite tree automata (SFTA) are FTA augmented with a weight function that attaches a weight to all accepting states.

Definition 7 (**SFTA**). *A state-weighted finite tree automaton (SFTA) over alphabet F is a tuple* $\mathcal{A} = (Q, F, Q_f, \Delta, w)$ *where Q is a set of states,* $Q_f \subseteq Q$ *is the set of accepting states,* Δ *is a set of transitions of the form* $f(q_1, \dots, q_k) \rightarrow q$ *where* $q, q_1, \dots q_k$ *are states,* $f \in F$ *and* $w : Q_f \rightarrow \mathbb{R}$ *is a function which assigns a weight* $w(q)$ *(from domain W) to each accepting state* $q \in Q_f$*.*

$$\frac{t \in T,\ \llbracket t \rrbracket \sigma = c}{q^c_t \in Q}\ (\textsc{Term}) \qquad \frac{q^c_{s_0} \in Q}{\begin{array}{c} q^c_{s_0} \in Q_f \\ w(q^c_{s_0}) = L(o, c) \end{array}}\ (\textsc{Final})$$

$$\frac{\begin{array}{c} s \rightarrow f(s_1, \dots s_k) \in P,\ \{q^{c_1}_{s_1}, \dots, q^{c_k}_{s_k}\} \subseteq Q, \\ \llbracket f(c_1, \dots c_k) \rrbracket \sigma = c \end{array}}{q^c_s \in Q,\ f(q^{c_1}_{s_1}, \dots, q^{c_k}_{s_k}) \rightarrow q^c_s \in \Delta}\ (\textsc{Prod})$$

Figure 4: Rules for constructing a SFTA $\mathcal{A} = (Q, F, Q_f, \Delta, w)$ given input σ, per-example loss function L, and grammar $G = (T, N, P, s_0)$.

Because CFTAs are designed to handle synthesis over clean (noise-free) data sets, they have only one accept state $q^o_{s_0}$ (the state with start symbol s_0 and output value o). We weaken this condition to allow multiple accept states with attached weights using SFTAs.

Given an input-output example (σ, o) and per-example loss function $L(o, c)$, Figure 4 presents rules for constructing a SFTA that, given a program p, returns the loss for p on example (σ, o). The SFTA Final rule (Figure 4) marks all states $q^c_{s_0}$ with start symbol s_0 as accepting states regardless of the concrete value c attached to the state. The rule also associates the loss $L(o, c)$ for concrete value c and output o with state $q^c_{s_0}$ as the weight $w(q^c_{s_0}) = L(o, c)$. The CFTA Final rule (Figure 3), in contrast, marks only the state $q^o_{s_0}$ (with output value o and start state s_0) as the accepting state.

A SFTA divides the set of programs in the DSL into subsets. Given an input σ, all programs in a subset produce the same output (based on the accepting state), with the SFTA assigning a weight $w(q^c_{s_0}) = L(o, c)$ as the weight of this subset of programs.

We denote the SFTA constructed from DSL G, example (σ, o), per-example loss function L, and threshold d as $\mathcal{A}^d_G(\sigma, o, L)$. We omit the subscript grammar G and threshold d wherever it is obvious from context.

Example 3. *Consider the DSL presented in Example 2. Given input-output example $(\{x \rightarrow 1\}, 9)$ and weight function $l(c) = (c - 9)^2$, Figure 5 presents the SFTA which represents all programs of height less than* 3.

For readability, we omit the states for terminals 2 and 3. For all accepting states the first number (the number in black) represents the computed value and the second number (the number in red) represents the weight of the accepting state.

6.1 Operations over SFTAs

Definition 8 (+ **Intersection**). *Given two SFTAs $\mathcal{A}_1 = (Q_1, F, Q^1_f, \Delta_1, w_1)$ and $\mathcal{A}_2 = (Q_2, F, Q^2_f, \Delta_2, w_2)$, a SFTA $\mathcal{A} = (Q, F, Q_f, \Delta, w)$ is the + intersection $\mathcal{A}_1$ and $\mathcal{A}_2$, if the CFTA in $\mathcal{A}$ is the intersection of CFTAs of $\mathcal{A}_1$ and $\mathcal{A}_2$, and the weight of accept states in $\mathcal{A}$ is the sum of weight of corresponding weights in $\mathcal{A}_1$ and $\mathcal{A}_2$. Formally:*

- *The CFTA (Q, F, Q_f, Δ) is the intersection of CFTAs $(Q_1, F, Q^1_f, \Delta_1)$ and $(Q_2, F, Q^2_f, \Delta_2)$*
- *$w(q^{\vec{c_1}:\vec{c_2}}_s) = w_1(q^{\vec{c_1}}_s) + w_2(q^{\vec{c_2}}_s)$ (for $q^{\vec{c_1}:\vec{c_2}}_s \in Q_f$).*

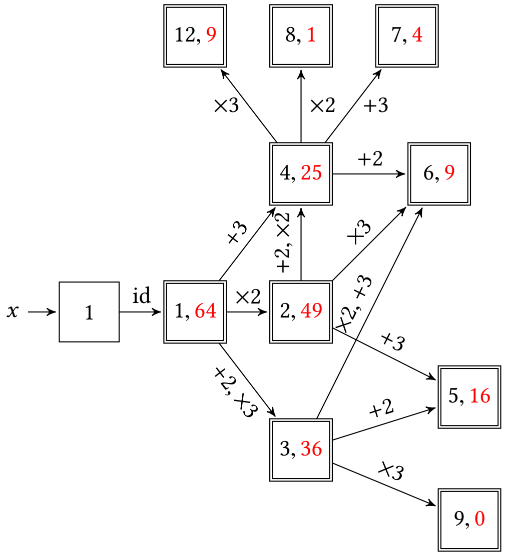

Figure 5: The SFTA constructed for Example 3

Given two SFTAs $\mathcal{A}_1$ and $\mathcal{A}_2$, $\mathcal{A}_1 + \mathcal{A}_2$ denotes the + intersection of $\mathcal{A}_1$ and $\mathcal{A}_2$.

Definition 9 (/ **Intersection**). *Given a SFTA $\mathcal{A} = (Q, F, Q_f, \Delta, w)$ and a CFTA $\mathcal{A}^* = (Q^*, F, Q^*_f, \Delta^*)$, a SFTA $\mathcal{A}' = (Q', F, Q'_f, \Delta', w')$ is the / intersection of $\mathcal{A}$ and $\mathcal{A}^*$, if the FTA $\mathcal{A}'$ is the intersection of CFTA $\mathcal{A}$ and $\mathcal{A}^*$, and the weight of the accepting state in $\mathcal{A}'$ is the same as the weight of the corresponding accepting state in $\mathcal{A}$. Formally:*

- *The CFTA (Q', F, Q'_f, Δ') is the intersection of FTAs (Q, F, Q_f, Δ) and $(Q^*, F, Q^*_f, \Delta^*)$*
- *$w'(q^{\vec{c_1}:\vec{c_2}}_s) = w(q^{\vec{c_1}:\vec{c_2}}_s)$ (for $q^{\vec{c_1}:\vec{c_2}}_s \in Q'_f$).*

Given a SFTA $\mathcal{A}$ and a CFTA $\mathcal{A}^*$, $\mathcal{A}/\mathcal{A}^*$ denotes the / intersection of $\mathcal{A}$ and $\mathcal{A}^*$.

Given a single input-output example, a CFTA built on that example only accepts programs which are consistent with that example. / intersection is a simple method to prune a SFTA to only contain programs which are consistent with an input-output example.

Definition 10 (w_0-**pruned SFTA**). *A SFTA $\mathcal{A}' = (Q, F, Q'_f, \Delta, w')$ is the w_0-pruned SFTA of $\mathcal{A} = (Q, F, Q_f, \Delta, w)$ if we remove all accept states with weights greater w_0 from Q_f. Formally, $Q'_f = \{q | q \in Q_f \wedge w_0 <= w(q)\}$ and $w'(q) = w(q)$ if $q \in Q'_f$.*

Given a SFTA $\mathcal{A}$, $\mathcal{A} \downarrow_{w_0}$ denotes the w_0-pruned SFTA of $\mathcal{A}$.

Definition 11 (q-**selection**). *Given a SFTA $\mathcal{A} = (Q, F, Q_f, \Delta, w)$ and a accept state $q \in Q_f$, the CFTA $(Q, F, \{q\}, \Delta)$ is called the q-selection of SFTA $\mathcal{A}$.*

Given a SFTA $\mathcal{A}$, the notation $\mathcal{A}_q$ denotes the q-selection of SFTA $\mathcal{A}$.

7 SYNTHESIS OVER NOISY DATA

Given a data set $\{(\sigma_1, o_1), \ldots, (\sigma_n, o_n)\}$ of input-output examples and loss function $\mathcal{L}(p, \mathcal{D})$ with per-example loss function L, we construct SFTAs for each input-output example $\mathcal{A}_1, \mathcal{A}_2, \ldots \mathcal{A}_n$ where $\mathcal{A}_i = \mathcal{A}(\sigma_i, o_i, L)$.

Theorem 1. *Given a SFTA $\mathcal{A} = \mathcal{A}(\sigma, o, L) = (Q, F, Q_f, \Delta, w)$, for all accepting states $q \in Q_f$ and for all programs p accepted by the q-selection automata $\mathcal{A}_q$:*

$$L(o, [\![p]\!]\sigma) = w(q)$$

Proof. Consider any state $q \in Q_f$. All programs accepted by state q compute the same concrete value c on the given input σ. Hence for all programs accepted by the q-selection automata $\mathcal{A}_q$, $[\![p]\!]\sigma = c$. By construction (Figure 4), $w(q) = L(c) = L([\![p]\!]\sigma)$ □

Let SFTA $\mathcal{A}(\mathcal{D}, L)$ be the + intersection of SFTAs defined on input-output examples in $\mathcal{D}$. Formally:

$$\mathcal{A}(\mathcal{D}, L) = \mathcal{A}(\sigma_1, o_1, L) + \mathcal{A}(\sigma_2, o_2, L) + \ldots \mathcal{A}(\sigma_n, o_n, L)$$

Since the size of each SFTA $\mathcal{A}(\sigma_i, o_i, L)$ is bounded, the cost of computing $\mathcal{A}(\mathcal{D}, L)$ is $O(|\mathcal{D}|)$.

Theorem 2. *Given $\mathcal{A}(\mathcal{D}, L) = (Q, F, Q_f, \Delta, w)$ as defined above, for all accepting states $q \in Q_f$, for all programs p accepted by the q-selection automata $\mathcal{A}(\mathcal{D}, L)_q$:*

$$\mathcal{L}(p, \mathcal{D}) = w(q)$$

i.e., the weight of the state q measures the loss of programs by q on data set $\mathcal{D}$.

Proof. Consider any accepting state $q \in Q_f$. Since $\mathcal{A}(\mathcal{D}, L)$ is an intersection of SFTAs $\mathcal{A}_1 \ldots \mathcal{A}_n$ (where $\mathcal{A}_i = \mathcal{A}(\sigma_i, o_i, L) = Q_i, F, (Q_f)_i, \Delta_i, w_i)$), there exist accepting states $q_1 \in (Q_f)_1, q_2 \in (Q_f)_2, \ldots q_n \in (Q_f)_n$ such that all programs p accepted by $\mathcal{A}_q$ are accepted by $(\mathcal{A}_1)_{q_1}, (\mathcal{A}_2)_{q_2} \ldots (\mathcal{A}_n)_{q_n}$. From Theorem 1, for all programs p accepted by $\mathcal{A}_q$, $w_i(q_i) = L(o_i, [\![p]\!]\sigma_i)$ From definition of + intersection,

$$w(q) = \sum_{i=1}^{n} w_i(q_i) = \sum_{i=1}^{n} L(o_i, [\![p]\!]\sigma_i) = \mathcal{L}(p, \mathcal{D})$$

□

Algorithm 1: Synthesis Algorithm

```
Input: DSL $G$, threshold $d$, data set $\mathcal{D}$, per-example loss
       function $L$, complexity measure $C$, and objective
       function $U$
Result: Synthesized program $p^*$
$\mathcal{A}(\mathcal{D}, L) = (Q, F, Q_f, \Delta, w)$
#the SFTA over data set $\mathcal{D}$ and per-example loss function $L$
foreach $q \in Q_f$ do
    $p_q \leftarrow \text{argmin}_{p \in \mathcal{A}(\mathcal{D}, L)_q} C(p)$
    # For each accepting state $q$, find the most optimal
      program $p_q$
end
$q^* \leftarrow \text{argmin}_{q \in Q_f} U(w(q), C(p_q))$
$p^* \leftarrow p_{q^*}$
```

Algorithm 1 presents the base algorithm to synthesize programs within various noisy synthesis settings.

Theorem 3. *The program p^* returned by Algorithm 1 is equal to p' where*

$$p' = \text{argmin}_{p \in G_d} U(\mathcal{L}(p, \mathcal{D}), C(p))$$

Proof. Given $\mathcal{A}(\mathcal{D}, L) = (Q, F, Q_f, \Delta, w)$. p^* returned by Algorithm 1 is equal to p_{q^*}, where $q^* = \text{argmin}_{q \in Q_f} U(w(q), C(p_q))$, where $p_q = \text{argmin}_{p \in \mathcal{A}(\mathcal{D}, L)_q} C(p)$. We can rewrite q^* as

$$\text{argmin}_{q \in Q_f} U(w(q), \min_{p \in \mathcal{A}(\mathcal{D}, L)_q} C(p))$$

Since for any l, $U(l, c)$ is non-decreasing with respect to c, we can rewrite q^* as

$$\text{argmin}_{q \in Q_f} \min_{p \in \mathcal{A}(\mathcal{D}, L)_q} U(w(q), C(p))$$

By Theorem 2, for any $p \in \mathcal{A}(\mathcal{D}, L)_q$:

$$w(q) = \mathcal{L}(p, \mathcal{D})$$

$$q^* = \text{argmin}_{q \in Q_f} \min_{p \in \mathcal{A}(\mathcal{D}, L)_q} U(\mathcal{L}(p, \mathcal{D}), C(p))$$

Because q^* is the accepting state of p^* and $p \in \mathcal{A}(\mathcal{D}, L)$ if and only if $\exists\, q \in Q_f . p \in \mathcal{A}(\mathcal{D}, L)_q$, we can rewrite the above equation as:

$$p^* = \text{argmin}_{p \in \mathcal{A}(\mathcal{D}, L)} U(\mathcal{L}(p, \mathcal{D}), C(p))$$

The set of programs accepted by $\mathcal{A}(\mathcal{D}, L)$ is the same set of programs in grammar G_d. Hence proved. □

We next present several modifications of the core algorithm to solve various synthesis problems.

7.1 Accuracy vs. Complexity Tradeoff

Given a DSL G, a data set $\mathcal{D}$, loss function $\mathcal{L}$, complexity measure c, and positive weight λ, we wish to find a program p^* which minimizes the weighed sum of the loss function and the complexity measure. Formally:

$$p^* = \text{argmin}_{p \in G_d} (\mathcal{L}(p, \mathcal{D}) + \lambda \cdot C(p))$$

where G_d is the set of programs in DSL G with size less than the threshold d. By using the objective function $U(l, c) = l + \lambda c$, we can use Algorithm 1 to synthesize program p^* which minimizes the objective function given above.

7.2 Best Program for Given Accuracy

Given a DSL G, a data set $\mathcal{D}$, loss function $\mathcal{L}$, complexity measure C and bound b, we wish to find a program p^* that minimizes the complexity measure C but has loss less than b. Formally: $p^* = \text{argmin}_{p \in G_d} C(p)$ s.t. $\mathcal{L}(p, \mathcal{D}) < b$. Note that this condition can be rewritten as

$$p^* = \text{argmin}_{p \in \mathcal{A}'} C(p)$$

where $\mathcal{A}' = \mathcal{A}(\mathcal{D}, L) \downarrow_b$.

By the definition of $\downarrow_b$, all accepting states of $\mathcal{A}'$ have weight less than b. Therefore all programs accepted by $\mathcal{A}'$ have loss less than b (i.e. $\mathcal{L}(p, \mathcal{D}) < b$). Also note that if a program p is not in $\mathcal{A}'$ then either it has loss greater than b or it is not within the threshold d.

7.3 Forced Accuracy

Given DSL G, a data set $\mathcal{D}$, a subset $\mathcal{D}' \subseteq \mathcal{D}$, loss function $\mathcal{L}$, complexity measure C, and objective function U, we wish to find a program p^* which minimizes the objective function with an added constraint of bounded loss over data set $\mathcal{D}'$. Formally:

$$p^* = \mathsf{argmin}_{p \in G_d} U(\mathcal{L}(p, \mathcal{D}), C(p)) \text{ s.t. } \mathcal{L}(p, \mathcal{D}') \leq b$$

We first construct a SFTA $\mathcal{A}(\mathcal{D}', L) \downarrow_b$ which contains all programs consistent with loss less than or equal to b over data set $\mathcal{D}'$. After constructing $\mathcal{A}(\mathcal{D}, L)$ as in Algorithm 1, we modify $\mathcal{A}(\mathcal{D}, L)$ by / intersection $\mathcal{A}(\mathcal{D}', L)$ (after dropping the weights of the accepting states) with $\mathcal{A}(\mathcal{D}, L)$ (i.e. $\mathcal{A}(\mathcal{D}, L) \leftarrow \mathcal{A}(\mathcal{D}, L) / \mathcal{A}(\mathcal{D}', L)$ as in Algorithm 1). By definition of / intersection and $\mathcal{A}$, loss of all programs returned by the modified algorithm on $\mathcal{D}'$ will be less than equal to b.

8 USE CASES

Definition 12. **Bayesian Program Synthesis***: Given a set of input-output examples $\mathcal{D} = \{(\sigma_i, o_i) | i = 1 \ldots n\}$, DSL grammar G, and a probability distribution π, p^* is the solution to the Bayesian program synthesis problem, if p^* is the most probable program in DSL G, given the data set $\mathcal{D}$. Formally $p^* = \mathsf{argmax}_{p \in G} \pi(p \mid \mathcal{D})$.*

By Bayes rule $p^* = \mathsf{argmax}_{p \in G} \pi(\mathcal{D}|p)\pi(p)$, so

$$p^* = \mathsf{argmax}_{p \in G} \Big[(\log \pi(\mathcal{D}|p)) + (\log \pi(p)) \Big]$$

Assuming independence of observations:

$$p^* = \mathsf{argmax}_{p \in G_d} \Big[\Big(\sum_{(\sigma_i, o_i) \in \mathcal{D}} \log \pi(o_i | [\![p]\!] \sigma_i) \Big) + (\log \pi(p)) \Big]$$

Where $\pi(o_i | [\![p]\!] \sigma_i)$ denotes the probability of output observation o_i in the data set $\mathcal{D}$, given a program p With complexity measure $\log \pi(p)$, per-example loss function $\log \pi(o_i | [\![p]\!] \sigma_i)$ (given example (σ_i, o_i)), and the following loss function:

$$\mathcal{L}(p, \mathcal{D}) = \sum_{(\sigma_i, o_i) \in \mathcal{D}} \log \pi(o_i | [\![p]\!] \sigma_i))$$

the technique in Section 7.1 (Algorithm 1) synthesizes the most probable program p^*

At Most k Wrong: Consider a setting in which, given a data set, a random procedure is allowed to corrupt at most k of these input-output examples. Given this noisy data set $\mathcal{D}$, our task is to synthesize the simplest program p^* which is wrong on at most k of these input-output examples. Formally, given data set $\mathcal{D}$, bound k, DSL G, and a complexity measure C:

$$p^* = \mathsf{argmin}_{p \in G} C(p) \text{ s.t. } \mathcal{L}_{0/1}(p, \mathcal{D}) \leq k$$

where $\mathcal{L}_{0/1}$ is the 0/1 loss function. The best program for a given accuracy framework (subsection 7.2) allows us to synthesize p^* subject to a threshold d.

9 EXPERIMENTAL RESULTS

String transformations have been extensively studied within the Programming by Example community [10, 13, 18]. We implemented our technique (in 6k lines of Java code) and used it to solve benchmark program synthesis problems from the SyGuS 2018 benchmark suite [2]. This benchmark suite contains a range of string transformation problems, a class of problems that has been extensively studied in past program synthesis projects [10, 13, 18].

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

Figure 6: DSL for string transformation, τ is a token, k is an integer, and s is a string constant

We use the DSL from [19] (Figure 6) with the size complexity measure Size(p). The DSL supports extracting and contatenating (Concat) substrings of the input string x; each substring is extracted using the SubStr function with a start and 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).

9.1 Implementation Optimizations

Instead of computing individual SFTAs for each input-output example, then combining the SFTAs via + intersections to obtain the final SFTA, our implementation computes the final SFTA directly working over the full data set. The implementation also applies two techniques that constrain the size of the final SFTA. First, it bounds the number of recursive applications of the production $e := \mathsf{Concat}(f, e)$ by applying a *bounded scope height threshold* d. Second, during construction of the SFTA, a state with symbol e is only added to the SFTA if the length of the state's output value is not greater than the length of the output string plus one.

9.2 Scalability

We evaluate the scalability of our implementation by applying it to all problems in the SyGuS 2018 benchmark suite [1]. For each problem we use the clean (noise-free) data set for the problem provided with the benchmark suite. We use the lexicographic objective function $U_L(l, c)$ with the $0/\infty$ loss function and the $c = \mathsf{Size}(p)$ complexity measure. We 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 SFTA. A state with symbol e is only added to the SFTA if the length of its output value is not greater than the length of the output string.

Because the running time of our technique does not depend on the specific utility function (except for the time required to evaluate the utility function, which is typically negligible for most utility functions, and except for search space pruning techniques appropriate for specific combinations of utility functions and DSLs), we anticipate that these results will generalize to other utility 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 a correct program does not exist within the DSL at bounded scope height threshold 4.

Table 7 presents results for selected SyGuS 2018 benchmarks. We omit all *-long-repeat benchmarks. We also omit all name-combine-4-*, phone-3-*, phone-4-*, phone-9-*, phone-10-*, and univ-* benchmarks — all runs for these benchmarks either do not synthesize a correct program or do not terminate. The full paper presents results for all SyGuS 2018 benchmarks.[1] There is a row for each benchmark

[1] The full paper is available at https://arxiv.org/abs/2009.10272.

Threshold	1		2		3		4	
Benchmark Name	time(sec)	SFTA size	time(sec)	SFTA size	time(sec)	SFTA size	time(sec)	SFTA 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-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-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-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-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-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-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-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-short	X	X	X	X	42.266	911.53	526.13	8104.7
reverse-name	X	X	6.9	269.9	217.19	7495.9	-	-
reverse-name-long	X	X	29.55	161.53	-	-	-	-
reverse-name-short	X	X	6.84	269.9	228.24	7485.8	-	-
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-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-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-short	0.09	0.46	0.47	1.58	0.83	1.58	2.78	1.58
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-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-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-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-short	0.27	0.64	0.97	2.6	2.45	2.61	13.81	2.61

Figure 7: Runtimes and SFTA sizes for selected SyGuS 2018 benchmarks

Benchmark	Data Set Size	Number of Required Correct Examples		
		1-Delete	DL	0/1
bikes	6	0	0	2
dr-name	4	0	0	1
firstname	4	0	0	2
lastname	4	0	2	4
initials	4	0	2	2
reverse-name	6	0	0	2
name-combine	4	0	0	2
name-combine-2	4	0	0	2
name-combine-3	4	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	2	3
phone-7	7	0	2	3
phone-8	7	0	0	1

Figure 8: Minimum number of correct examples required to synthesize correct a program.

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 SFTA (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 SFTA increase as the number of provided input-output examples and/or the bounded height threshold increases. The SFTA size sometimes stops increasing as the height threshold increases. We attribute this phenomenon to the application of a search space pruning technique that terminates the recursive application of the production $e := \text{Concat}(f, e)$; 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 data set.

We compare with a previous technique that uses FTAs to solve program synthesis problems [20]. This previous technique requires clean data and only synthesizes programs that agree with all input-output examples in the data set. Our technique builds SFTAs with similar structure, with additional overhead coming from the evaluation of the objective function. We obtained the implementation of the technique presented in [20] and ran this implementation on all benchmarks in the SyGuS 2018 benchmark suite. The running times of our implementation and this previous implementation are comparable.

9.3 Noisy Data Sets, Character Deletion

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

We exclude all *-long, *long-repeat, and *-short benchmarks and all benchmarks that do not terminate within the time limit at height bound 4. For each remaining benchmark we use our implementation and the generated corrupted data sets to determine the minimum number of correct outputs in the corrupted data set required for the implementation to produce a correct program that matches the original hidden clean data set on all input-output examples. We consider three loss functions: the 0/1 and DL loss functions (Section 3) 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 13. **1-Delete Loss Function:** *The 1-Delete loss function $\mathcal{L}_{1D}(p, \mathcal{D})$ uses the per-example loss function L that is 0 if the outputs from the synthesized program and the data set match exactly, 1 if a single deletion enables the output from the synthesized program to match the output from the data set, and ∞ otherwise:*

$$\mathcal{L}_{1D}(p, \mathcal{D}) = \sum_{(\sigma_i, o_i) \in \mathcal{D}} L_{1D}(\llbracket p \rrbracket \sigma_i, o_i), \text{ where}$$

$$L_{1D}(o_1, o_2) = \begin{cases} 0 & o_1 = o_2 \\ 1 & a \cdot x \cdot b = o_1 \wedge a \cdot b = o_2 \wedge |x| = 1 \\ \infty & \text{otherwise} \end{cases}$$

We use the lexicographic objective function $U_L(l, c)$ with $c = \text{Size}(p)$ as the complexity measure and bounded scope height threshold $d = 4$. We apply a search space pruning technique that terminates the recursive application of the production $e := \text{Concat}(f, e)$; when the generated string becomes more than one character longer than the current output string.

Table 8 summarizes the results. The Data Set Size Column presents the total number of input-output examples in the corrupted data set. 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 data set on all input-output examples) using the 1-Delete, DL, and 0/1 loss functions, respectively.

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 data set even when given a data set 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 data set 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 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, sometimes encounters incorrect programs that fit the corrupted data better, and therefore sometimes requires correct inputs to synthesize the correct program.

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.

9.4 Noisy Data Sets, Character Replacements

We next present results for our 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 data sets for these benchmarks contain 400 input-output examples, including repeated input-output examples.

For each of these phone-*-long-repeat benchmark problems on which our technique terminates with bounded scope height threshold 4 (Section 9.2), we construct a noisy data set as follows. For each digit in each output string in the data set, we flip a biased coin. With probability b, we 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 $9/10 \times b$).

We then run our implementation on each benchmark problem with the noisy data set using the tradeoff objective function $U_\lambda(l, c) = l + \lambda \times c$ with complexity measure $c = \text{Size}(p)$ and the following n-Substitution loss function:

Definition 14. ***n*-Substitution Loss Function:** *The n-Substitution loss function $\mathcal{L}_{nS}(p, \mathcal{D})$ uses the per-example loss function L_{nS} that 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 data set, the loss function is ∞:*

$$\mathcal{L}_{nS}(p, \mathcal{D}) = \sum_{(\sigma_i, o_i) \in \mathcal{D}} L_{nS}([\![p]\!]\sigma_i, o_i), \ \textit{where}$$

$$L_{nS}(o_1, o_2) = \begin{cases} \infty & |o_1| \neq |o_2| \\ \sum\limits_{i=1}^{|o_1|} 1 \ \textit{if}\ o_1[i] \neq o_2[i] \ \textit{else}\ 0 & |o_1| = |o_2| \end{cases}$$

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 9: Approximate program synthesis with DL loss.

We 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 data set.

These results highlight, once again, the ability of our 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.

9.5 Approximate Program Synthesis

For the benchmarks in Table 9, a correct program does not exist within the DSL at bounded scope threshold 2. Table 9 presents results from our implementation on the clean (noise-free) benchmark data sets with the DL loss function, Size(p) complexity measure, lexicographic objective function $U_L(\mathcal{L}_{DL}(p, \mathcal{D}), \text{Size}(p))$, 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 data set, the DL loss incurred by the synthesized program over the entire data set, the sum of the lengths of the output strings of the data set (the DL loss for an empty output would be this sum), and the size of the synthesized program.

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 our technique to approximate a correct program when the correct program does not exist in the program search space.

9.6 Discussion

Practical Applicabilty: The experimental results show that our 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 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 data sets 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 data sets in which a significant fraction of the outputs are corrupted. We 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.

We note that our 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 system of which we are 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, our results provide encouraging evidence that our technique can help program synthesis systems work effectively with noisy data sets. Important future work in this area will more fully 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 data sets, as well as a meaningful evaluation of program synthesis more generally, awaits this future work.

Noise Sources With Different Characteristics: Our 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 our technique to extract and exploit this preserved information to enhance the effectiveness of the synthesis. The question may arise how well may our 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 we would expect the algorithm to require a balance of correct outputs before it would be able to synthesize the correct program.

10 RELATED WORK

The problem of learning programs from a set of input-output examples has been studied extensively [10, 16, 18]. These techniques can be largely broken down into the following four categories:

Synthesis Using Solvers: These systems require the user to provide precise semantics for the operators for the DSL they are using [11]. Our technique, in contrast, only requires black-box implementations of these operators. A large class of these systems depend on solvers which do not scale as the number of examples increases. Since our techniques are based on tree automata, our cost linearly increases as the number of examples increase. These systems require all input-output examples to be correct and only synthesize programs that match all input-output examples.

Enumerative Techniques: These techniques search the space of programs to find a single program that is consistent with the given examples [8, 12]. 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 [12]. These techniques require all input-output examples to be correct and only synthesize programs that match all input-output examples.

VSA-based/Tree Automata-based Techniques: These techniques build complex data structures representing all possible programs compatible with the given examples [13, 18, 20]. Current work requires users to provide correct input-output examples. Our work modifies these techniques to handle noisy data and to synthesize approximate programs that minimize an objective function over the provided potentially noisy data set.

Neural Program Synthesis/ML Approaches: There is extensive work that uses machine learning/deep neural networks to synthesize programs [3, 6, 15]. These techniques require a training phase and a differentable loss function. Our technique requires no training phase and can work with arbitrary loss functions including, for example, the 0/1 loss function. Machine learning techniques are incompatible with this type of loss function. These systems provide no guarantees over the completeness and the optimality of their result, whereas our technique, due to its property of exploring all programs of size less than a threshold, always finds a program within the bounded scope that minimizes the objective function.

Data Set Sampling or Cleaning: There has been recent work which aspires to clean the data set or pick representative examples from the data set for synthesis [10, 14, 15], for example by using machine learning or data cleaning to select productive subsets of the data set over which to perform exact synthesis. In contrast to these techniques, our proposed techniques 1) provide deterministic guarantees (as opposed to either probabilistic guarantees as in [15] or no guarantees at all as in [10, 14]), 2) do not require the use of oracles as in [15], 3) can operate successfully even on data sets 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 data set to drive the synthesis as in [10, 15].

Active Learning: Recent research exploits the availability of a reference implementation to use active learning for program synthesis [17]. Our technique, in contrast, works directly from given input-output examples with no reference implementation.

11 CONCLUSION

Dealing with noisy data is a pervasive problem in modern computing environments. Previous program synthesis systems target data sets in which all input-output examples are correct to synthesize programs that match all input-output examples in the data set.

We present a new program synthesis technique for working with noisy data and/or performing approximate program synthesis. Using state-weighted finite tree automata, this technique supports the formulation and solution of a variety of new program synthesis problems involving noisy data and/or approximate program synthesis. The results highlight how this technique, by exploiting information from a variety of sources — structure from the underlying DSL, information left intact by discrete noise sources — can deliver effective program synthesis even in the presence of substantial noise.

ACKNOWLEDGMENTS

This research was supported, in part, by the Boeing Corporation and DARPA Grants N6600120C4025 and HR001120C0015.

REFERENCES

[1] 2018. SyGuS 2018 String Benchmark Suite. https://github.com/SyGuS-Org/benchmarks/tree/master/comp/2019/PBE_SLIA_Track/from_2018. Accessed: 2020-07-18.

[2] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. IEEE, 1–8.

[3] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989* (2016).

[4] Christopher M Bishop. 2006. *Pattern recognition and machine learning*. springer.

[5] Fred J. Damerau. 1964. A Technique for Computer Detection and Correction of Spelling Errors. *Commun. ACM* 7, 3 (March 1964), 171âĂŞ176. https://doi.org/10.1145/363958.363994

[6] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. Robustfill: Neural program learning under noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 990–998.

[7] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 422–436.

[8] John K Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 229–239.

[9] Giorgio Gallo, Giustino Longo, Stefano Pallottino, and Sang Nguyen. 1993. Directed hypergraphs and applications. *Discrete applied mathematics* 42, 2-3 (1993), 177–201.

[10] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *ACM Sigplan Notices*, Vol. 46. ACM, 317–330.

[11] Shachar Itzhaky, Sumit Gulwani, Neil Immerman, and Mooly Sagiv. 2010. A simple inductive synthesis methodology and its applications. In *ACM Sigplan Notices*, Vol. 45. ACM, 36–46.

[12] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. *ACM SIGPLAN Notices* 50, 6 (2015), 619–630.

[13] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 107–126.

[14] Yewen Pu, Zachery Miranda, Armando Solar-Lezama, and Leslie Pack Kaelbling. 2017. Selecting representative examples for program synthesis. *arXiv preprint arXiv:1711.03243* (2017).

[15] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. 2016. Learning programs from noisy data. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 761–774.

[16] D Shaw. 1975. Inferring LISP Programs From Examples.

[17] Jiasi Shen and Martin C Rinard. 2019. Using active learning to synthesize models of applications that access databases. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 269–285.

[18] Rishabh Singh and Sumit Gulwani. 2016. Transforming spreadsheet data types using examples. In *Acm Sigplan Notices*, Vol. 51. ACM, 343–356.

[19] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Program synthesis using abstraction refinement. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 63.

[20] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Synthesis of data completion scripts using finite tree automata. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 62.

[21] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. 2016. Synthesizing transformations on hierarchically structured data. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 508–521.