Inductive Program Synthesis Guided by Observational Program Similarity

JOHN FESER, MIT, USA
ISIL DILLIG, UT Austin, USA
ARMANDO SOLAR-LEZAMA, MIT, USA

We present a new general-purpose synthesis technique for generating programs from input-output examples. Our method, called *metric program synthesis*, relaxes the *observational equivalence* idea (used widely in bottom-up enumerative synthesis) into a weaker notion of *observational similarity*, with the goal of reducing the search space that the synthesizer needs to explore. Our method clusters programs into equivalence classes based on an expert-provided *distance metric* and constructs a version space that compactly represents "approximately correct" programs. Then, given a "close enough" program sampled from this version space, our approach uses a distance-guided repair algorithm to find a program that exactly matches the given input-output examples. We have implemented our proposed metric program synthesis technique in a tool called SyMetric and evaluate it in three different domains considered in prior work. Our evaluation shows that SyMetric outperforms other domain-agnostic synthesizers that use observational equivalence and that it achieves results competitive with domain-specific synthesizers that are either designed for or trained on those domains.

ACM Reference Format:
John Feser, Isil Dillig, and Armando Solar-Lezama. 2018. Inductive Program Synthesis Guided by Observational Program Similarity. *Proc. ACM Program. Lang.* 1, CONF, Article 1 (January 2018), 29 pages.

1 INTRODUCTION

Programming-by-example (PBE) is a program synthesis task where the goal is to learn a program in some domain-specific language (DSL) that is consistent with a set of input-output examples. Because PBE can automate custom tasks without requiring programming skills, this topic has attracted enormous attention from several research communities and has found many useful applications ranging from string and table transformations [Feng et al. 2017; Gulwani 2011] to question answering [Chen et al. 2021] to computer-aided design (CAD) [Du et al. 2018].

Techniques for solving the PBE problem can be classified as either *domain-agnostic* or *domain-specific*. Domain-specific methods (e.g., [Chen et al. 2021, 2020; Du et al. 2018; Feng et al. 2017; Feser et al. 2015; Gulwani 2011; Wang et al. 2017a]) are specialized to a DSL and target a pre-defined class of synthesis tasks. Domain-agnostic methods [Feng et al. 2018; Solar-Lezama et al. 2006; Wang et al. 2018] are parameterized over a DSL and can, in principle, be applied to a variety of domains.

A common domain-agnostic solution to the PBE problem is to perform *bottom-up enumeration* over DSL programs [Albarghouthi et al. 2013; Miltner et al. 2022; Udupa et al. 2013; Wang et al. 2018]. The idea is to start with primitives in the DSL and build up increasingly complex programs by combining existing terms via DSL constructs. The key challenge when scaling this approach to large programs is state explosion, since the number of programs grows exponentially with the search depth. Prior work has proposed several techniques to combat the state explosion problem. One simple but effective technique is to leverage *observational equivalence*: programs that produce the same output on the given set of input examples are effectively identical (at least for PBE purposes), so it suffices to keep only one representative program. For example, synthesis techniques based on

Authors' addresses: John Feser, MIT, CSAIL, 32 Vassar St, Cambridge, USA, feser@mit.edu; Isil Dillig, UT Austin, Austin, USA, isil@cs.utexas.edu; Armando Solar-Lezama, MIT, CSAIL, 32 Vassar St, Cambridge, USA, asolar@mit.edu.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

finite tree automata (FTA) leverage observational equivalence to compactly represent of the set of all programs consistent with the given input-output examples.

Observational equivalence only reduces the state space in scenarios where many programs share the same (relevant) input-output behavior, but this property does not hold in all domains. Consider the *inverse constructive solid geometry (CSG) problem*, where the goal is to "de-compile" a complex geometric shape into a set of geometric operations that were used to construct it in a computer aided design (CAD) system [Du et al. 2018; Nandi et al. 2018; Shapiro and Vossler 1991; Willis et al. 2021]. While this problem can be framed as a PBE task [Du et al. 2018], observational equivalence only modestly reduces the search space because few programs produce *exactly* the same image. However, we notice that, in this domain, many programs have similar—but not identical—input-output behaviors. Furthermore, replacing a subprogram with one that has similar behavior often causes a small change to the overall program behavior, which suggests that these replacements can be repaired. This observation motivates the following question: can we relax the observational equivalence criterion and develop a synthesis algorithm that exploits our domain knowledge about the *semantic similarity* between programs?

In this paper, we answer this question affirmatively and present a new PBE algorithm called *metric program synthesis*. We generalize observational equivalence to a weaker criterion called *observational similarity* by replacing the equivalence relation with an expert-provided distance function δ. Our method clusters programs into the same equivalence class if their output is within a radius ϵ according to δ. This distance metric gives DSL designers a powerful way to inject domain knowledge into the search without building an entirely domain-specific algorithm. While this approach makes assumptions about the properties of the DSL, we give sufficient conditions for completeness (§3.6) and we show empirically that our algorithm improves performance even on DSLs that do not fully satisfy these conditions.

To exploit observational similarity, our metric program synthesis approach proceeds in two phases: First, it performs bottom-up enumerative synthesis to build a *version space* [Lau et al. 2003] that represents all programs up to some fixed AST depth. During bottom-up enumeration, it clusters programs into equivalence classes using the provided distance metric, keeping one representative of each equivalence class. Distance-based clustering introduces approximation: the version space contains many programs that are incorrect but *close to* being correct. We introduce a second *local search* step to repair these incorrect programs: starting with a program P whose output is close to the goal, our technique performs hill-climbing search guided by δ to find a syntactic perturbation P' of P that has the intended input-output behavior. For DSLs that are sufficiently continuous, this step mitigates the incompleteness introduced by distance-based clustering and allows recovering programs that are not explicitly contained in the reduced search space.

We implemented our approach in a tool called SyMetric and evaluate it on three domains: (1) inverse CSG [Du et al. 2018], (2) regular expression synthesis [Chen et al. 2020; Lee et al. 2016], and (3) tower building [Ellis et al. 2021]. Our evaluation shows that, when provided with appropriate distance functions, SyMetric is competitive with synthesizers designed/trained for these domains and it outperforms other general-purpose synthesizers that use observational equivalence.

To summarize, this paper makes the following contributions:

- We introduce *metric program synthesis* as a way to generalize observational equivalence and give DSL designers a new mechanism for injecting domain knowledge into the synthesizer.
- We show how to use distance metrics to perform effective clustering, ranking, and repair of programs explored during synthesis.
- We evaluate our implementation, SyMetric, in three application domains and compare it against several relevant baselines, including a large language model.

2 OVERVIEW

In this section, we work through an example that illustrates the key ideas in our algorithm.

Consider the picture of a key shown in Fig. 1. To a human, it is clear that this image contains three important pieces: circles to make up the handle of the key, a rectangle for the shaft, and evenly spaced rectangles for the teeth. We can write a program that generates this key in a simple DSL that includes primitive circles and rectangles as well as union, difference, and repetition operators:

$$(\texttt{Circle}(4,8,4) - \texttt{Circle}(4,8,3)) \cup \texttt{Rect}(7,7,15,9) \cup \texttt{Repeat}(\texttt{Rect}(10,9,11,10),2,0,3)$$

The program composes the three shapes: a hollow circle for the handle of the key, a rectangle for the shaft, and three small, evenly spaced rectangles for the teeth. The hollow circle is constructed by subtracting a small circle from a larger one: $\texttt{Circle}(x,y,r) - \texttt{Circle}(x',y',r')$. The evenly spaced teeth are constructed by replicating a small rectangle: $\texttt{Repeat}(\texttt{Rect}(x,y,x',y'),dx,dy,3)$. Circles are specified by a center point and radius; rectangles are specified by their lower left and upper right corners.

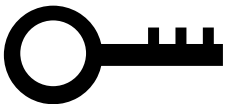

Fig. 1. The input image.

Suppose that our goal is to synthesize the program above given *just* the picture in Fig. 1. As mentioned in §1, a standard approach is to perform bottom-up search over programs in the DSL: i.e. create programs by composing together smaller terms, but discard those that create a previously-seen shape. This approach—known as bottom-up enumeration with equivalence reduction [Udupa et al. 2013]—is a simple, powerful domain-agnostic synthesis algorithm that works in many domains.

However, the inverse CSG domain is full of programs that are *similar*, but not identical. To see why, consider the two circles that make up the handle of the key. If the outer circle was slightly larger or slightly shifted, it would still be clear to us that the image is only slightly perturbed. We would be able to fix the program by locally improving it—i.e., shifting the circle back into place by changing its parameters. We should not need to retain both programs in the search space, since one transforms straightforwardly into the other. However, we cannot use equivalence reduction to group these two programs together, even though our intuition tells us that they should be nearly interchangeable.

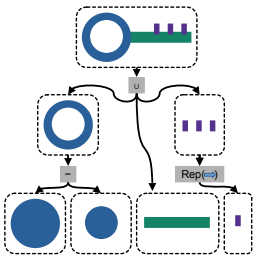

Fig. 2. The solution, illustrated.

To synthesize the figure above, our algorithm proceeds in two phases. It first performs coarse-grained search to look for a program P that is *close to* matching the target image. Then it applies perturbations to P to find a repair that *exactly* matches the given image. We now explain these two phases in detail.

Global coarse-grained search: The first phase of our algorithm is based on bottom-up search and, like prior work [Wang et al. 2018], it builds a data structure that compactly represents a large space of programs. We represent the space of programs using a variant of a finite tree automaton (FTA) called an *approximate finite tree automaton* (XFTA) (§3.3). The key idea is to group together values that are semantically similar: in the CSG context, images that are sufficiently similar to each other are represented using the same state in the automaton.

We construct this XFTA in three phases: *expansion*, *grouping*, and *ranking*. During expansion, operators are applied to sub-programs to create new candidate programs. For our running example, the first expansion step generates the set of primitive shapes. Later expansions compose shapes together using set operators and the `Repeat` construct to create images of increasing complexity.

In the grouping phase, images are clustered. Each cluster has a center c and radius ϵ, and every image in the cluster is within distance ϵ of c. Although every image in the cluster is retained as

Fig. 3. A sketch of the local search process for the key example.

part of the search space, only the center of the cluster participates in further expansion steps. The clustering phase is essentially a relaxed version of equivalence reduction.

Finally, in the ranking phase, the w clusters that are closest to the goal image are retained, with the goal of focusing search on the programs that are likely to produce the goal. After ranking, the top w clusters are inserted as new states into the XFTA, and the operators that produced each state in the cluster are inserted as edges.

When the global search terminates, the XFTA represents a space of programs that are close to the target image but it may not contain exactly the target image. To address this issue, our method performs a second level of *local search*.

Local fine-grained search: The local search proceeds in two phases: it extracts a candidate program from the XFTA, and it attempts to repair the candidate.

Since each node in the XFTA represents a (possibly) exponentially large set of programs, we use a greedy algorithm to select a program from this set rather than attempting to exhaustively search over it. Starting at an accepting state of the automaton, the program extractor selects the incoming edge that produces the image closest to the goal. Selecting an edge determines the root operator of the candidate program and the program sets from which to select arguments. Extraction proceeds recursively, always minimizing the distance between the overall candidate program and the goal.

When a candidate program has been extracted, we attempt to *repair* it by applying syntactic rewrites. The sequence of rewrites is chosen by a form of tabu search [Glover and Laguna 1998] and is guided by the distance from the candidate program to the goal. At each step of the repair process, we consider the set of programs obtained by applying a single rewrite rule to the current program and choose the closest to the goal. This process continues until the desired program is found or until a maximum number of rewrites have been applied.

Figure 3 gives a high-level view of this repair process. Starting from a program whose output is similar to the input image, the repair process applies rewrites such as changing squares to circles or incrementing/decrementing numeric parameters. Each rewrite gets us closer to the target image so the local search can often quickly converge to a program that exactly produces the target.

3 METRIC PROGRAM SYNTHESIS ALGORITHM

In this section, we describe our proposed synthesis algorithm. Given a language L and a set of input-output examples of the form $\{(I_1, O_1), \ldots, (I_n, O_n)\}$, our goal is to synthesize a program P in L such that $\forall i \in [1, n]$. $[\![P]\!](I_i) = O_i$, meaning that evaluating P on I_i yields O_i for every example.

The rest of this section is organized as follows: First, because our method builds on bottom-up synthesis using finite tree automata, we start with preliminary information on FTAs in §3.1. Then, in §3.2, we describe our top-level synthesis algorithm, followed by discussions of its three sub-procedures in §3.3 and §3.5.

3.1 Background on Synthesis using FTAs

Our synthesis algorithm builds on prior work on synthesis using finite tree automata (FTA). At a high level, an FTA is a generalization of a DFA from words to trees; just as a DFA accepts words, an FTA recognizes trees. FTAs are defined as follows:

Definition 3.1. **(FTA)** A bottom-up finite tree automaton (FTA) over alphabet Σ is a tuple $\mathcal{A} = (Q, Q_f, \Delta)$ where Q is the set of states, $Q_f \subseteq Q$ are the final states, and Δ is a set of transitions of the form $\ell(q_1, \ldots, q_n) \rightarrow q$ where $q, q_1, \ldots, q_n \in Q$ and $\ell \in \Sigma$.

FTAs are useful in synthesis because they can compactly encode a set of programs represented by their abstract syntax trees [Miltner et al. 2022; Wang et al. 2017b, 2018]. When used for synthesis, states of the FTA correspond to values (e.g., integers), and the alphabet corresponds to the set of DSL operators (e.g., $+, \times$). Final states are marked based on the specification, and transitions model the semantics of the underlying DSL. For instance, in a language with a negation operator $\neg$, transitions $\neg(0) \rightarrow 1$ and $\neg(1) \rightarrow 0$ express the semantics of negation.

We can view terms over an alphabet Σ as trees of the form $T = (n, V, E)$ where n is the root node, V is a set of labeled vertices, and E is the set of edges. A term T is said to be accepted by an FTA if T to can be rewritten to some state $q \in Q_f$ using transitions Δ. Finally, the language of a tree automaton $\mathcal{A}$ is denoted as $\mathcal{L}(\mathcal{A})$ and consists of the set of all terms accepted by $\mathcal{A}$.

Given a specification φ, the idea behind FTA-based synthesis is to build an FTA whose language is the set of all programs satisfying φ. FTAs can be used to solve PBE problems as follows: For each input-output example (I, O), start with a state representing I and construct new states and transitions by applying the DSL operators. For example, given FTA states representing integers 1 and 2 and a + operator in the DSL, we generate a new state representing 3 using the transition $+(1, 2) \rightarrow 3$. This process of adding new states and transitions to the FTA continues until either there are no more states to be added or a predefined bound on the number of states or transitions has been reached. The final state of the FTA corresponds to the output example O and the language of the FTA includes programs that are consistent with the example. Multiple examples can be handled either by generating a FTA for each example and intersecting them or by generating a single FTA that represents all of the examples. In this work, we use the latter strategy.

As this discussion makes clear, an FTA-based approach can compactly represent the version space if many DSL programs share the *same* input-output behavior (because such programs lead to the same FTA state). FTA-based synthesis may not scale in application domains that do not have this property. Prior work on FTA-based synthesis has tried to tackle this problem using abstract interpretation and abstraction refinement [Wang et al. 2018]: in that setting, FTA states correspond to *abstract* rather than *concrete* values, and transitions are constructed using the *abstract* semantics of the DSL. Since an *abstract FTA* over-approximates the set of programs consistent with the specification, one needs to perform abstraction refinement to iteratively rule out spurious programs from the language of the FTA. While this so-called Syngar approach has proven to be effective in some domains like tensor manipulations [Wang et al. 2018], such abstractions are not always easy to construct. For example, we have found the inverse CSG domain to *not* be amenable to an abstract interpretation approach. Our metric-based synthesis algorithm is an attempt to solve this problem in a different way using distances rather than abstract domains.

3.2 Overview of Synthesis Algorithm

As mentioned earlier, the key idea behind metric-based synthesis is to relax the *observational equivalence* criterion into *observational similarity* by using a distance metric. More formally, we define observational similarity as follows:

Definition 3.2. **(Similarity)** Two values v and v' are *similar*, denoted $v \simeq_\epsilon v'$ if they are within ϵ of each other, according to a distance metric δ:

$$v \simeq_\epsilon v' \equiv \delta(v, v') \leq \epsilon.$$

Algorithm 1 Metric synthesis algorithm.

Require: Σ is a set of operators, I and O are the input and output examples respectively, c_{max} is the maximum program size to consider when constructing the XFTA, w is the beam width, δ is a distance metric between values, ϵ is the threshold for clustering.

Ensure: On success, returns a program p where $[\![p]\!] = O$. On failure, returns $\perp$.

```
1: procedure MetricSynth(Σ, I, O, c_max, w, δ, ε)
2:     A ← ConstructXFTA(Σ, I, O, c_max, w, δ, ε)
3:     for P ∈ Extract(A, I, O, q, δ) do
4:         P ← Repair(I, O, δ, P)
5:         if P ≠ ⊥ then return P
6:     return ⊥
```

The main idea behind our synthesis algorithm is to construct an approximate version space by clustering together values that are similar. Just as the Syngar idea of [Wang et al. 2018] groups together values based on an *abstract* notion of observational equivalence, our method groups together values based on this notion of *similarity* and constructs a so-called *approximate FTA (XFTA)* representing programs that produce values close to the desired output.

Our top-level metric program synthesis approach is presented in Alg. 1 and is parameterized over a (1) distance metric δ, (2) radius ϵ, and (3) domain-specific language L. At a high level, this algorithm consists of three steps:

(1) **XFTA construction:** First, MetricSynth constructs an FTA (line 2 of Alg. 1) that represents a space of programs that produce values close to the goal. However, because this FTA is constructed by grouping similar values together, a program excepted by this automaton does not necessarily satisfy the specification.

(2) **Program extraction:** To deal with the approximation introduced by clustering, the algorithm enters a loop in which it repeatedly extracts programs from the FTA using the call to Extract at line 3. The goal of Extract is to find a program in the language of the FTA that produces a value that is sufficiently close to the target.

(3) **Program repair:** Because the extracted program does not satisfy the input-output examples in the general case, the Repair procedure (invoked at line 4) tries to find a syntactic perturbation of P that exactly satisfies the input-output examples. As we discuss in more detail in §3.5, the repair procedure is based on rewrite rules and performs a form of tabu search, using the distance metric as a guiding heuristic.

We now discuss the ConstructXFTA, Extract, and Repair procedures in detail.

3.3 Approximate FTA Construction

Algorithm 2 shows our technique for constructing an approximate FTA for a given set of input-output examples. At a high level, this algorithm builds programs in a bottom-up fashion, clustering together those programs that produce similar values on the same input. To ensure that the algorithm terminates, it builds programs up to a fixed depth controlled by the hyperparameter c_{max}.

In more detail, ConstructXFTA adds new automaton states and transitions (initialized to I and $\emptyset$ respectively) in each iteration of the while loop. For each (n-ary) DSL operator ℓ and existing states $q_1, \ldots, q_n$, it gets a new *frontier* of candidate transitions $\Delta_{frontier}$ by evaluating $\ell(q_1, \ldots, q_n)$. The construction of this frontier corresponds to the *expansion phase* mentioned in §2.

The expansion phase can produce many new states, making XFTA construction prohibitively expensive. Thus, in the next *clustering phase* (line 5 of Alg. 2), the algorithm groups similar states

Algorithm 2 Algorithm for constructing an approximate FTA.

Require: Σ is a set of operators, all other parameters are the same as in Algorithm 1. k is a hyper-parameter that determines the number of states that the automaton should accept.
Ensure: Returns an XFTA.

1: **procedure** ConstructXFTA($\Sigma, I, O, c_{max}, w, \delta, \epsilon$)
2: $Q \leftarrow I, \Delta \leftarrow \emptyset$
3: **for** $1 \leq c \leq c_{max}$ **do**
4: $\Delta_{frontier} \leftarrow \{\ell(q_1, \ldots, q_n) \rightarrow q \mid \ell \in \Sigma, \{q_1, \ldots, q_n\} \subseteq Q, [\![\ell(q_1, \ldots, q_n)]\!] = q\}$
5: $(Q_c, \Delta_c) \leftarrow$ Cluster($\Delta_{frontier}, \delta, \epsilon$)
6: $Q' \leftarrow$ TopK($Q_c, \delta(O), w$)
7: $Q \leftarrow Q \cup Q', \quad \Delta \leftarrow \Delta \cup \{(\ell(q_1, \ldots, q_n) \rightarrow q) \mid q \in Q', (\ell(q_1, \ldots, q_n) \rightarrow q) \in \Delta_c\}$
8: $Q_f \leftarrow$ TopK($Q, \delta(O), k$)
9: **return** (Q, Q_f, Δ)

Algorithm 3 Greedy algorithm for clustering states.

Require: Δ is a set of FTA transitions, all other parameters are the same as in Algorithm 1.
Ensure: Returns a set of FTA transitions.

1: **procedure** Cluster(Δ, δ, ϵ)
2: $Q' \leftarrow \emptyset, \Delta' \leftarrow \emptyset$ ▷ New (clustered) states and transitions
3: **for** $(\ell(q_1, \ldots, q_n) \rightarrow q) \in \Delta$ **do**
4: $close \leftarrow \{q_{center} \in Q' \mid \delta(q_{center}, q) \leq \epsilon\}$
5: **if** $close = \emptyset$ **then** $close \leftarrow \{q\}$
6: $Q' \leftarrow Q' \cup close, \quad \Delta' \leftarrow \Delta' \cup \{\ell(q_1, \ldots, q_n) \rightarrow q' \mid q' \in close\}$
7: **return** (Q', Δ')

introduced by expansion into a single state (Alg. 3). Standard clustering algorithms like k-means are not suitable in this context because they fix the number of clusters but allow the radius of each cluster to be arbitrarily large. Instead, we would like to minimize the number of clusters while ensuring that the radius of each cluster is bounded. Hence, we use the Cluster procedure from Alg. 3 to generate a set of clusters where each state is within some ϵ distance from the center of a cluster. To do so, Alg. 3 iterates over the new states q in the frontier and starts a new cluster for q if none of the previous frontier states are within ϵ of q (lines 4–5 in Alg. 3). Otherwise, q is added to an existing cluster (line 6 in Alg. 3). For each new transition $\ell(q_1, \ldots, q_n) \rightarrow q$ of the frontier, clustering produces new transitions of the form $\ell(q_1, \ldots, q_n) \rightarrow q_c$ where q_c is the center of a cluster that q belongs to. Clustering produces a new set of states Q_c and a new set of transitions Δ_c to add to the automaton (line 5 of Alg. 2).

The final step in XFTA construction is the *ranking phase* (lines 6–8 of Alg. 2). Even after clustering, the automaton might end up with a prohibitively large number of new states, so ConstructXFTA only keeps the top w clusters in terms of their distance to the goal. Thus, in each iteration, Alg. 2 only ends up adding w new states to the automaton, like beam search.

3.4 Extracting Programs from XFTA

We now turn our attention to the Extract procedure for picking a program that is accepted by our approximate FTA. Recall that programs accepted by the XFTA are not necessarily consistent with the input-output examples due to clustering. Furthermore, two programs P, P' that are accepted by

Algorithm 4 Algorithm for extracting programs from an XFTA.

Require: $\mathcal{A}$ is an XFTA; all other parameters are the same as in Algorithm 1.
Ensure: Yields program terms that are accepted by $\mathcal{A}$.

```
1:  procedure Extract(A, I, O, δ)
2:      Q_f ← FinalStates(A)
3:      Sort Q_f by δ(O) increasing.
4:      for q_f ∈ Q_f do
5:          Δ_root ← {(ℓ(q_1, ..., q_n) → q_f) | (ℓ(q_1, ..., q_n) → q_f) ∈ Transitions(A)}
6:          yield ExtractTerm(Δ_root, I, δ(O))

7:  procedure ExtractTerm(Δ, I, δ)
8:      Let (ℓ(q_1, ..., q_n) → q) ∈ Δ be a transition where q minimizes δ(q)
9:      for 1 ≤ i ≤ n do                    ▷ Extract a program for each argument to ℓ.
10:         δ_i ← λq. δ([[ℓ(q'_1, ..., q'_{i-1}, q, q_{i+1}, ..., q_n)]])
11:         p_i ← ExtractTerm(Δ, I, δ_i)
12:         q'_i ← [[p_i]](I)
13:     return ℓ(p_1, ..., p_n)
```

the XFTA need not be equally close to the goal state; for example, $[\![P]\!](I)$ might be much closer to O than $[\![P']\!](I)$ according to the distance metric δ. Ideally, we would like to find the best program that is accepted by the FTA (in terms of its proximity to the goal); however, this can be prohibitively expensive, as the automaton (potentially) represents an exponential space of programs. Thus, rather than finding the best program accepted by the automaton, our Extract procedure greedily selects a sequence of "good enough" programs in a computationally tractable way.

The high-level idea behind Extract is to recursively build a program starting from a final state q_f via the call to the recursive procedure ExtractTerm. At every step, the algorithm picks a transition $\ell(q_1, \ldots, q_n) \rightarrow q$ whose output minimizes the distance from the goal and then recursively constructs the arguments $p_1, \ldots, p_n$ of ℓ. Note that this algorithm is greedy in the sense that it tries to find a single operator that minimizes the distance from the goal rather than a sequence of operators (i.e., the whole program). Hence, there is no guarantee that Extract will return the optimal program accepted by $\mathcal{A}$.

3.5 Distance-Guided Program Repair

The final part of our synthesis algorithm (Repair) takes the program that was extracted from the XFTA and attempts to repair it by applying syntactic rewrite rules. In particular, given a program P that is close to the goal, Repair tries to find a program P' that is (1) syntactically close to P and (2) correct with respect to the input-output examples (i.e., $[\![P]\!](I) = O$).

Our Repair procedure is parameterized by a set of rewrite rules R of the form $t \rightarrow s$. We say that a program P can be rewritten into P' if there is a rule $r = (t \rightarrow s) \in R$ and a substitution σ such that $P = \sigma t$ and $P' = \sigma s$. We denote the application of rewrite rule r to P as $P \rightarrow_r P'$.

The Repair procedure is presented in Alg. 5 and applies goal-directed rewriting to the candidate program, using the distance function δ to guide the search. In particular, it starts with the input program P and iteratively applies a rewrite rule until either a correct program is found or a bound n on the number of rewrite rules is reached. In each iteration of the loop (lines 3–6), it first generates a set of new candidate programs (called *neighbors*) by applying a rewrite rule to P and (greedily)

picks the program P' that minimizes the distance $\delta(O, [\![P']\!](I))$. In the next iteration, the new program P' is used as the seed for applying rewrite rules.

Algorithm 5 Algorithm for repairing a program.

Require: I, O are the input output examples, δ is a distance metric, P is a program. There are also two hyperparameters: n is the maximum number of rewrites to perform, and R is a set of rewriting rules.

Ensure: Returns a program P' such that $[\![P']\!](I) = O$ or returns $\perp$.

1: **procedure** REPAIR(I, O, δ, P)
2: $S \leftarrow \emptyset$
3: **while** $i < n$ **do**
4: $P \leftarrow \arg\min_{p \in neighbors} \delta(O, [\![p]\!](i))$ **where** $neighbors = \{P' \mid P \rightarrow_r P', r \in R\} - S$
5: **if** $[\![P]\!](I) = O$ **then return** P
6: $S \leftarrow S \cup \{P\}$
7: **return** $\perp$

Note that our REPAIR procedure uses a (bounded) set S to avoid getting stuck in local minima, as in tabu search [Glover and Laguna 1998]. The set S contains the most recently explored k programs, and, when applying a rewrite rule, the REPAIR procedure avoids generating any program in S. Despite this mitigation, REPAIR is a local search; we do not expect it to find a global minimum from any point in the search space. By building the XFTA and performing local search on all final states, we get to search in many different basins that could contain a solution.

For REPAIR to make progress, there are some constraints on the choice of rewrite rules and on the DSL operators. The rewrite rules need to be able to incrementally improve programs, which means that there needs to be a sequence of locally reachable programs between a candidate and the goal. The operators also need to have a "transparency property", which means that local changes in their arguments should be visible in their results (i.e. there are no magic parameters which give no feedback until they are correctly set). We discuss these constraints in the next section.

METRICSYNTH should continue to work with other implementations of REPAIR (e.g. gradient based) as long as they satisfy the conditions in §3.6.

3.6 Theoretical Guarantees

As stated earlier, our synthesis algorithm does not come with completeness guarantees *in general*, but it does so under certain assumptions. While these conditions are fairly strong and not likely to be met in many application domains of interest, we believe stating these conditions is useful for successfully instantiating metric synthesis in new domains.

First, we give notation for local search (§3.5) reachability. Let $P \rightsquigarrow P'$ denote that $\exists P''$. $[\![P']\!] = [\![P'']\!] \wedge \text{REACH}(P, P'')$, which says that a program equivalent to P' is reachable from P via local search. We say that $P \leftrightsquigarrow P'$ iff $P \rightsquigarrow P' \wedge P' \rightsquigarrow P$. We note that hill climbing search is generally symmetric (and is for our domains); if the local search is symmetric, then $P \rightsquigarrow P' \vee P' \rightsquigarrow P \implies P \leftrightsquigarrow P'$.

Assumption #1. First, for our theoretical analysis, we assume that the distance function and local search are related. We formalize this relationship using the notion of *local reachability*:

Definition 3.3. **(Local reachability)** The *local reachability* property requires:

$$[\![P]\!] \simeq_\alpha [\![P']\!] \implies P \leftrightsquigarrow P'$$

Intuitively, this property states that programs that are within some small distance α from each other can be rewritten to one another through a small number of applications of the rewrite rules.

Hence, the local reachability property depends on the chosen set of rewrite rules for the target application domain.

Assumption #2. Our second assumption is the so-called *directionality* property, which, intuitively, states that subprograms of the target program should *not* be prohibitively far away from the goal:

Definition 3.4. **(Directionality)** Let $P^\star$ be the desired solution for the synthesis task. The monotonicity assumption requires:

$$\forall P, C.\ \delta([\![P]\!], [\![P^\star]\!]) > \beta \implies C[P] \not\leftsquigarrow P^\star.$$

This definition says that if a program P is far enough from the goal $P^\star$, then it is not a useful subprogram; there is no context C that we can place it in that is in the neighborhood of $P^\star$. If this property does not hold, then our algorithm may prune useful subprograms from the search space.

Assumption #3. Finally, we assume that all DSL operators preserve the ability to find programs with local search. We refer to this property as *transparency*:

Definition 3.5. **(Transparency)** The transparency property requires that, for every n'ary DSL operator f, we have:

$$\bigwedge_{i=1}^{n} a_i \leftsquigarrow b_i \implies f(a_1, \ldots, a_n) \leftsquigarrow f(b_1, \ldots, b_n)$$

Along with local reachability, this assumption justifies the use of grouping, because it says that retaining the center of a group suffices to recover the rest of the programs in the group.

Under these assumptions, we can state the following completeness property of metric synthesis:

Theorem 3.6. *Let $P^\star$ be a program in a language L, consistent with I/O examples (I, O), where $|P^\star| < c_{max}$, and where the beam width w is large enough that all programs within β of $P^\star$ are retained. Under the three assumptions discussed above,* MetricSynth$(L, I, O, c_{max}, w, \delta, \alpha/2)$ *will return a program equivalent to $P^\star$.*

To prove the completeness theorem, we first prove the following lemma about the XFTA produced by ConstructXFTA. We denote "a state q accepts a program P" as $P \in q$. We say that for $\mathcal{A} = (Q, Q_f, \Delta)$, a program equivalent to $P^\star$ is reachable from $\mathcal{A}$ iff $\exists q \in Q_f . \forall P \in q.\ P \leftsquigarrow P^\star$, and we denote this as $\mathcal{A} \overset{*}{\rightsquigarrow} P^\star$.

Lemma 3.7. *ConstructXFTA$(\Sigma, I, O, c_{max}, w, \delta, \alpha) = \mathcal{A}$ and for all subterms P of $P^\star$, $\mathcal{A} \overset{*}{\rightsquigarrow} P$ (under the same assumptions as in Theorem 3.6).*

Proof. The proof is by induction on the size c of the largest programs in $\mathcal{A}$, following the loop in ConstructXFTA.

Base case. Let P be a subterm of $P^\star$ s.t. $|P| = 1$. We show that $\mathcal{A}_1 \overset{*}{\rightsquigarrow} P$. After building $\Delta_{frontier}$, we discard transitions $\ell \to q$ that are not in top-k. For a suitably chosen w, the top-k transitions are within distance β from the goal, so we have $\delta(q, [\![P^\star]\!]) > \beta$. By directionality, the transition labeled by P is retained because there is a context C s.t. $C[P] = P^\star$. We then cluster the transitions into groups of radius at most $\alpha/2$. By local reachability, P is reachable from any of the programs in its group. Therefore, $\mathcal{A}_1 \overset{*}{\rightsquigarrow} P$.

Inductive case. Let $P = a \oplus b$ be a subterm of $P^\star$ s.t. $|P| = c$. We assume a binary operator for simplicity of presentation. By induction, $\mathcal{A}_{c-1} \overset{*}{\rightsquigarrow} a$ and $\mathcal{A}_{c-1} \overset{*}{\rightsquigarrow} b$. Let q_a and q_b be the representative states of a and b.

We now show that the transition labeled by $\oplus(q_a, q_b)$ is retained after ranking. Let $a' \in q_a$ and $b' \in q_b$. By induction, $a' \leftsquigarrow a$ and $b' \leftsquigarrow b$. P is a subterm of $P^\star$, so there is some C s.t. $C[P] =$

$$E := \texttt{Circle}(x, y, r) \mid \texttt{Rect}(x_1, y_1, x_2, y_2) \mid E \cup E \mid E - E \mid \texttt{Repeat}(E, x, y, c).$$

$$M = \{(u, v) \mid 0 \le u < x_{max}, 0 \le v < y_{max}\}$$

$$\mathcal{E}[\texttt{Circle}(x, y, r)] = \left\{(u, v) \mid \sqrt{(x-u)^2 + (y-v)^2} < r, (u, v) \in M\right\}$$

$$\mathcal{E}[\texttt{Rect}(x_1, y_1, x_2, y_2)] = \{(u, v) \mid x_1 \le u \wedge y_1 \le v \wedge u \le x_2 \wedge v \le y_2, (u, v) \in M\}$$

$$\mathcal{E}[\texttt{Repeat}(E, x, y, c)] = \{(u, v) \mid (u - ix, v - iy) \in \mathcal{E}[E], 0 \le i < c, (u, v) \in M\}$$

$$\mathcal{E}[e \cup e'] = \mathcal{E}[e] \cup \mathcal{E}[e'] \qquad \mathcal{E}[e - e'] = \mathcal{E}[e] \setminus \mathcal{E}[e']$$

Fig. 5. The syntax and semantics of CSG programs. The semantics is given as an evaluation function.

$C[a \oplus b] = P^\star$. By transparency, $C[a' \oplus b'] \leftsquigarrow C[a \oplus b]$, so by directionality, $\delta([\![P]\!], [\![P^\star]\!]) \le \beta$, which means that the transition is retained. After clustering, P is reachable from any program in its group, by local reachability. Therefore, $\mathcal{A}_c \overset{*}{\rightsquigarrow} P$. □

Observe that the completeness theorem follows immediately from this lemma: Let $\mathcal{A}$ be the result of ConstructXFTA$(\Sigma, I, O, c_{max}, w, \delta, \alpha/2)$. For each final state q in $\mathcal{A}$, we extract a program $P \in q$ and perform local search. By the above lemma, there must be some q such that for any P, $P \leftsquigarrow P^\star$. Therefore, we always find a program equivalent to $P^\star$.

4 INSTANTIATING METRIC SYNTHESIS IN APPLICATION DOMAINS

The MetricSynth algorithm can be instantiated in different application domains by supplying a suitable DSL, distance metric, and rewrite rules for repair. In this section, we discuss how to instantiate it in inverse CSG, regular expression synthesis, and tower-building domains.

4.1 Instantiation for Inverse CSG

The *inverse CSG problem* aims to "de-compile" a complex geometric shape into a set of geometric operations that were used to construct it. We now describe how we use metric program synthesis framework to solve the inverse CSG problem.

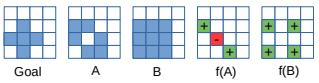

Fig. 4. Illustration of the distance transformation f_O. The Jaccard distance is $\delta_J(A, B) = \frac{1}{3}$ but $\delta_{Goal}(A, B) = \frac{3}{5}$. The distance between images that are very close to the goal is magnified.

4.1.1 Domain-Specific Language. Figure 5 shows the syntax and semantics of our CSG DSL. This DSL includes two primitive shapes: circles and rectangles. Circles are represented by a center coordinate and a radius. Rectangles are axis-aligned and are represented by the coordinates of the lower left and upper right corners. The primitive shapes can be combined using union, difference, and repeat operators. $\texttt{Repeat}(E, x, y, c)$ takes an image E, a translation vector $v = (x, y)$, and a count c, and it produces the union of E repeated c times, translated by v. For example: $\texttt{Repeat}(\texttt{Circle}(v, r), v', 2) = \texttt{Circle}(v, r) \cup \texttt{Circle}(v + v', r)$. `Repeat` allows programs with repeating patterns to be expressed compactly, which also makes these programs easier to synthesize.

We present the semantics of our CSG DSL using an evaluation procedure; $\mathcal{E}$ takes a program and produces a bitmap image. Bitmap images are represented as the set of all filled pixels (u, v).

4.1.2 Synthesis Problem. Given a bitmap image B of size $w \times h$, inverse CSG aims to synthesize a program E such that:

$$\forall 0 \le x < w, 0 \le y < h.\ B(x, y) \iff (x, y) \in \mathcal{E}[E]$$

Note that inverse CSG is exactly a programming-by-example (PBE) problem: We can think of the bitmap image as a set of I/O examples where each input example is a pixel (x, y) and the output example is a Boolean. However, a key difference from standard PBE is that the number of examples we need to deal with is quite large: for instance, in our evaluation, we use 32×32 bitmap images, so there are 1024 I/O examples.

4.1.3 Distance Function. The distance metric is a key component of our algorithm. For inverse CSG, we use a slight modification of the Jaccard distance that takes into account the goal value:

$$\delta_O(q, q') = 1 - \frac{|f_O(q) \cap f_O(q')|}{|f_O(q) \cup f_O(q')|} \quad \text{where } f_O(q) = \{(x, y, b) \mid (x, y) : b \in q, b \neq O[x, y]\}.$$

Intuitively, δ_O only considers the pixels that *differ* from the goal image O. This is desirable because it amplifies small differences between images that are close to the goal, as illustrated in Fig. 4. The original Jaccard distance does not satisfy local reachability (Def. 3.3), because there are programs whose output is largely similar, but differ in many small details that require significant transformation to add. We started with the Jaccard distance and changed it in response to the observation that the search was clustering programs that were not locally reachable (see Fig. 11). While this distance also does not satisfy local reachability in general, it gets closer as the search progresses and the space fills with programs that are similar to the output.

The distance function is predictive of the value of a subprogram to the solution as long as that subprogram does not appear on the right-hand-side of a subtraction.

4.1.4 Rewrite Rules. Recall that our Repair procedure is parameterized over a set of rewriting rules R. The rules we use for inverse CSG modify integers (within bounds) and transform squares into circles (and vice-versa):

$$\begin{aligned}
x &\longrightarrow x + 1 && \text{if } x \text{ is an integer and } x < x_{max} \\
x &\longrightarrow x - 1 && \text{if } x \text{ is an integer and } x > 0 \\
\texttt{Circle}(x, y, r) &\longrightarrow \texttt{Rect}(x - r, y - r, x + r, y + r) \\
\texttt{Rect}(x_1, y_1, x_2, y_2) &\longrightarrow \texttt{Circle}(x_1 + r, y_1 + r, r) && \text{where } r = \frac{x_2 - x_1}{2} \text{ if } x_2 - x_1 = y_2 - y_1
\end{aligned}$$

The circle-to-square rule in particular helps align the local search with the distance function in accordance with local reachability. Similarly sized and positions circles and squares are similar according to δ, so we ensure they are close according to the local search.

4.2 Instantiation for Regular Expressions

Our second application domain of metric program synthesis is generating regular expressions from a set of positive and negative examples. We chose this domain because it has a state-of-the-art program synthesizer [Chen et al. 2020] and because it is not a perfect fit for our technique. That our algorithm works on this domain is evidence that it may generalize to other domains as well.

4.2.1 Domain-Specific Language. Our regular expression language is a subset of the regular expression language used in prior work [Chen et al. 2020] (the DSL from prior work includes several operations that can be implemented in this subset). The syntax is:

$$\begin{aligned}
E := \emptyset \mid C \mid \texttt{Concat}(E, E) \mid \texttt{Repeat}(E, x) \mid \texttt{RepeatRange}(E, x_1, x_2) \mid \texttt{RepeatAtLeast}(E, x) \\
\mid \texttt{Optional}(E) \mid E \wedge E \mid E \vee E \mid \neg E
\end{aligned}$$

The DSL includes character classes, concatenation, repetition, optional matches, conjunction, disjunction, and negation. Character classes match a single character from a set; we use a set of

$$\mathcal{E}_s[C] = \{(i, i+1) \mid 0 \le i \le |s|, s[i] \in C\} \qquad \mathcal{E}_s[\emptyset] = \{(i,i) \mid 0 \le i \le |s|\}$$
$$\mathcal{E}_s[\texttt{Repeat}(E,1)] = \mathcal{E}_s[E] \qquad \mathcal{E}_s[\texttt{Repeat}(E,x)] = \mathcal{E}_s[\texttt{Concat}(E, \texttt{Repeat}(E, x-1))]$$
$$\mathcal{E}_s[\texttt{Optional}(E)] = \mathcal{E}_s[E \vee \emptyset] \qquad \mathcal{E}_s[\neg E] = \{(i,j) \mid 0 \le i \le j \le |s|, (i,j) \notin \mathcal{E}_s[E]\}$$
$$\mathcal{E}_s[E \wedge E'] = \mathcal{E}_s[E] \cap \mathcal{E}_s[E'] \qquad \mathcal{E}_s[E \vee E'] = \mathcal{E}_s[E] \cup \mathcal{E}_s[E']$$

$$\mathcal{E}_s[\texttt{RepeatRange}(E, x_1, x_2)] = \bigcup_{x_1 \le i \le x_2} \mathcal{E}_s[\texttt{Repeat}(E, i)]$$
$$\mathcal{E}_s[\texttt{RepeatAtLeast}(E, x)] = \mathcal{E}_s[\texttt{RepeatRange}(E, x, |s|)]$$
$$\mathcal{E}_s[\texttt{Concat}(E, E')] = \{(i,k) \mid (i,j) \in \mathcal{E}_s[E], (j',k) \in \mathcal{E}_s[E'], j = j'\}$$

Fig. 6. Semantics of the regular expression DSL. $\mathcal{E}_s[E]$ returns the positions in s that E matches.

single-character classes that includes all the printable characters as well as multi-character classes for numbers, capital and lowercase letters, symbols, and vowels. $\texttt{Concat}(E, E')$ matches E followed by E'. For example, Concat(<num>, <a>) matches the string "0a." $\texttt{Repeat}(E, x)$ matches x repetitions of E. Programs in this DSL evaluate to a set of match locations in a string s. The semantics of the DSL are shown in Figure 6.

4.2.2 Synthesis Problem. As in prior work [Chen et al. 2020; Lee et al. 2016], we consider the problem of synthesizing regular expressions from a given set of positive and negative examples. Let S^+ be a set of positive examples (strings), and let S^- be a set of negative examples. Then, the synthesis problem is to generate a regular expression E such that:

$$\forall s \in S^+.\ (0, |s|) \in \mathcal{E}_s[E] \text{ and } \forall s \in S^-.\ (0, |s|) \notin \mathcal{E}_s[E]$$

However, prior work has shown that synthesizing the *intended* regular expression just from positive and negative examples can be challenging if one only has access to a few examples. For this reason, [Chen et al. 2020] has advocated using *sketches* obtained from natural language descriptions. Following that work, we consider a modified version of this problem where the regular expression needs to be a completion of the provided sketch *and* satisfy the examples. Specifically, our problem formulation requires a sketch given in the form of a program with holes, where the holes may contain constraints on the terms that must be used to fill the hole, as in [Chen et al. 2020].

4.2.3 Distance Function. The distance function for regular expressions is the Jaccard distance between match sets, which means that expressions that match at the same positions are considered similar. This is predictive of the value of a subprogram to the overall solution unless that subprogram must be negated.

4.2.4 Rewrite Rules. Some of the constructs used in the regex DSL take integers as arguments. As in the inverse CSG instantiation, we include rewrite rules that transform integers. We also include a rule that rewrites Repeat to RepeatRange, because δ treats some programs that use Repeat as similar to programs that use RepeatRange, so the local search needs to reflect that property.

The operators in the regex DSL are largely transparent, in that a small change to the behavior of their arguments is reflected as a small change in their output. The least transparent operators are conjunction and disjunction.

4.2.5 Sketch Constraints. The regular expression synthesis problem is based on sketches, so our synthesizer must produce programs that satisfy the examples *and* match the sketch. Sketches consist of terms with holes $?\{E_1, \ldots, E_n\}$; holes have a list of associated *constraints*. These constraints are

$$\textsc{Top}(bs, x) = \max\{y \mid (b, x', y) \in bs, x = x'\}$$

$$\mathcal{E}[\texttt{DropV}, (h, bs)] = (h, (h, \textsc{Top}(h, bs), \texttt{V}) : bs) \qquad \mathcal{E}[\texttt{DropH}, (h, bs)] = (h, (h, \max_{h \le x < h+3} \textsc{Top}(x, bs), \texttt{H}) : bs)$$

$$\mathcal{E}[\texttt{MoveBefore}(E, x), (h, bs)] = \mathcal{E}[E, (h + x, bs)] \qquad \mathcal{E}[\texttt{MoveAfter}(E, x), s] = (h + x, bs) \text{ if } (h, bs) = \mathcal{E}[E, s]$$

$$\mathcal{E}[E; E', s] = \mathcal{E}[E', \mathcal{E}[E, s]] \qquad \mathcal{E}[\texttt{Embed}(E), (h, bs)] = (h, bs') \text{ if } (h', bs') = \mathcal{E}[E, s]$$

$$\mathcal{E}[\texttt{Loop}(1, E), s] = \mathcal{E}[E, s] \qquad \mathcal{E}[\texttt{Loop}(x, E), s] = \mathcal{E}[\texttt{Loop}(x - 1, E), \mathcal{E}[E, s]]$$

Fig. 7. Semantics of the tower building DSL.

terms; a term E matches a hole $?\{E_1, \ldots, E_n\}$ if one of $E_1, \ldots, E_n$ is a subterm of E. For example, <num> and ¬<num> both match ?{<num>}. Chen et al. discuss the semantics of sketches in detail.

We treat sketches as additional DSL operators—a sketch with k holes becomes an operator that takes k parameters. The regex DSL semantics are extended so that programs return a set of the sketch holes that they match in addition to the set of regex matches. The sketch operators check that their arguments match the appropriate holes; if not, they return a special error value. Programs that match distinct sets of sketch holes are treated as infinitely far apart, so are never clustered. This constrains the search space to only contain programs that conform to the sketch.

4.3 Instantiation for Tower Building

Our third application domain is the tower building task from prior work [Ellis et al. 2021; Nye et al. 2021] that is inspired by AI planning tasks. Given a set of blocks and target "tower" (i.e. configuration of these blocks), this task aims to generate the desired tower by (programmatically) controlling a robot arm.

We chose this domain because it has a strong baseline for comparison [Nye et al. 2021]. Some of the other domains from Ellis et al. may be appropriate our technique, but DreamCoder is primarily a library learner, not a synthesizer, so it is not an appropriate baseline.

4.3.1 Domain-Specific Language. The syntax of the tower building DSL is:

$$E := \texttt{DropH} \mid \texttt{DropV} \mid \texttt{MoveBefore}(E, x) \mid \texttt{MoveAfter}(E, x) \mid E; E \mid \texttt{Loop}(x, E) \mid \texttt{Embed}(E).$$

Semantics are shown in Figure 7. Programs in this language control a robot arm which can move left and right along a horizontal track and can drop horizontal or vertical blocks. The state of the program includes the x-position of the arm and the list of dropped blocks. The DSL includes operators for dropping blocks, moving the robot arm, sequencing, and looping. `DropH` and `DropV` both add a new (horizontal or vertical) block to the tower. The block will be placed on the highest block that is below the arm. The move operators both update the position of the arm; `MoveBefore` moves the arm and then executes E, while `MoveAfter` moves the arm after evaluating E. `Embed` gives the language a degree of modularity. It executes E and then resets the arm to wherever it was before. `Loop` repeats the body x times. Note that these semantics correspond to an idealized model where blocks fall in a straight line until they land on top of another block and blocks cannot topple.

4.3.2 Synthesis Problem. The input to the synthesizer is a set of blocks B. Each block is represented as a tuple (b, x, y) where $b \in \{\texttt{H}, \texttt{V}\}$ is the type of block (either horizontal 1×3 or vertical 3×1) and (x, y) is the block's position. B must be a valid tower, which means that the blocks must not overlap. The synthesis problem is to produce a program E such that $\mathcal{E}[E, s_0] = B$, where $s_0 = (0, [\,])$ is the initial state with no blocks placed and the hand at $x = 0$.

4.3.3 Distance Function. The distance function for the tower building domain is based on the insight that translating a tower along the x-axis is straightforward, so we want our distance function to be *translation-invariant*. Hence, two programs that build the same tower in nearby places should be deemed similar. Based on this intuition, we use the Jaccard distance to compare two towers, and we normalize them before we compare them so that their leftmost block is at $x = 0$:

$$\delta(s, s') = \delta_J(z(s), z(s')) \text{ where } z((h, bs)) = (h, \{(b, x - x_{min}, y) \mid (b, x, y) \in bs\})$$

where δ_J is the Jaccard distance and z is the normalizing function. This distance function is compatible with local reachability, because translating a tower is often as simple as changing the parameter to a move operator. It is also compatible with directionality, because it decreases the distance between the solution and programs that construct the right tower at the wrong place or construct only the lower part of a taller tower.

4.3.4 Rewrite Rules. As in the other domains, some of the constructs in the tower building DSL take integer valued arguments. Hence, we use rewrite rules that allow incrementing and decrementing these integers, as in the inverse CSG and regular expression domains. These rules suffice to change loop iteration counts and to modify the movement operators.

5 IMPLEMENTATION

We have implemented our synthesis technique in a new tool called SyMetric[1] written in OCaml. Our algorithm is implemented as a library that is parameterized over a DSL.

5.1 Optimizations

Our implementation includes some optimizations of the algorithm in §3.

Randomization. Our implementation of Repair considers a random subset of the rewrites when generating candidate programs to select from. This randomization compensates for the greedy nature of this algorithm by introducing the possibility of taking a locally suboptimal step that turns out to be globally optimal.

Incremental clustering. Since the clustering technique is a significant cost of approximate FTA construction, our implementation performs a few modifications. In particular, instead of computing all clusters and then sorting them, it first sorts the transitions and uses the first k clusters that it finds. Furthermore, because the number of transitions in $\Delta_{frontier}$ can be very large in the ConstructXFTA algorithm, our implementation incrementally collects the top states in batches. This involves evaluating the frontier multiple times, rather than storing it, but we find that, in practice, we need only a small prefix of the sorted frontier. Finally, our implementation of the Cluster procedure uses an M-tree data structure [Ciaccia et al. 1997] to facilitate efficient insertion and range queries.

Optimizations for inverse CSG. Our instantiation of SyMetric in the inverse CSG domain incorporates three low-level optimizations. First, it represents images as packed bitvectors to reduce their size. Second, our evaluation function for the CSG DSL is memoized. Third, our implementation uses optimized, vectorized ISPC [Pharr and Mark 2012] implementations for bitvector operations, distance functions, and for CSG operators such as `Repeat`.

Optimizations for regular expressions. In our implementation of the regular expression domain we view the match sets as graphs where the positions in the string are the nodes and the matches are the edges. We represent these graphs as adjacency matrices using an efficient packed Boolean representation. This representation is particularly effective for synthesizing regular expressions

[1] https://github.com/jfeser/symetric

from examples, because the example strings tend to be short, which mitigates the $O(n^2)$ memory cost of the matrix. We also note that `Repeat`(E, n) matches (i, j) if E matches $(i, k_1), (k_1, k_2), \ldots, (k_{n-1}, j)$. That is, `Repeat`$(E, n)$ matches (i, j) if there is a walk of length n in the match graph for E from i to j. The walks of length n are given by the nth power of the adjacency matrix A_E^n. Therefore, we can build efficient implementations for the `Repeat*` operators and for `Concat` using an efficient Boolean matrix multiplication primitive.

5.2 Hyperparameters

Algorithm 1 takes three hyperparameters: the number of rewrites n, the number of clusters of each cost w, and the cluster radius ϵ. In general, increasing n and w and decreasing ϵ makes the search more expensive by increasing the size of the search space and increases the probability of finding a solution.

Due to grouping, the XFTA accumulates error as the search proceeds—the distance between cluster centers and the programs in the cluster increases. The ϵ and w parameters may be used to control this error; decreasing ϵ will reduce error by making the clusters smaller but w will need to be increased to retain the same space of programs, which will increase the size of the XFTA. There is a fundamental tradeoff between error accumulation and XFTA size.

6 EVALUATION

In this section, we describe a series of experiments to empirically evaluate our approach. In particular, our experiments are designed to evaluate the following key research questions:

- **RQ1:** How does SyMetric compare against other domain-agnostic and domain-specific synthesis tools in the inverse CSG, regular expression, and tower building domains?
- **RQ2:** What is the relative importance of the various ideas comprising our approach?
- **RQ3:** How do different components of our synthesis algorithm contribute to running time?

6.1 Inverse CSG

In our evaluation on the inverse CSG domain, we compare SyMetric against the following baselines:

- Sketch-Du: We implement a baseline using the Sketch synthesis system and an encoding like the one used in InverseCSG [Du et al. 2018]. However, since that work focuses on 3D shapes, we modify the encoding to work with our DSL for 2D geometry and also add support for repetition (see §7). Unlike the encoding in [Du et al. 2018], our sketches do not contain hints about the location of primitive shapes, so the synthesizer needs to discover the numeric parameters of these primitive shapes.
- FTA: This baseline performs bottom-up synthesis (with equivalence reduction) using FTAs [Wang et al. 2017b]. Like the implementation of SyMetric, this baseline is also implemented in OCaml. Note that this baseline only adds FTA states and transitions until a final state is reached, as our goal is to find *one*, rather than all, programs consistent with the specification.
- AFTA: This baseline implements the Syngar algorithm for bottom up synthesis with abstract FTAs (AFTAs) [Wang et al. 2018]. Syngar uses abstract values as states of the FTA, constructs FTA transitions using the abstract semantics, and performs abstraction refinement to deal with spurious programs extracted from the AFTA. Our implementation of this baseline is also in OCaml and uses the matrix abstract domain from [Wang et al. 2018] since bitmap images can be viewed as matrices.

Benchmarks. While prior work on inverse CSG [Du et al. 2018; Jones et al. 2021] mostly targets 3D benchmarks, we construct our own benchmark suite for 2D inverse CSG. We chose to work in the 2D setting because our original motivation for this work was to generalize the InverseCSG

Fig. 8. Generated and handwritten CSG benchmarks.

algorithm [Du et al. 2018] to remove domain specific preprocessing and problem partitioning and to support the setting where looping constructs need to be discovered in order to capture regularities in a design. The CSG-specific preprocessing step that InverseCSG employs produces a restricted space of geometry primitives that are then combined using Sketch. Removing this preprocessing step makes the problem significantly harder, because Sketch must search a richer space of programs. We found that the 3D problems are too challenging for the Sketch baseline without the preprocessing, but the 2D version of the problems are tractable enough to allow for a meaningful comparison.

We consider a total of 47 benchmarks (Fig. 8), where 25 correspond to outputs of randomly generated programs (modulo some non-triviality constraints) and 22 are handwritten benchmarks of visual interest. The handwritten programs have between 9 and 35 AST nodes and the generated programs have between 20 and 30.

Setup. We run all our experiments on a machine with two AMD EPYC 7763 processors (256 threads total) and 1TB of RAM. We use a time limit of 1 hour and a memory limit of 4 GB (so that we can run many benchmarks at the same time). For the hyperparameters for SyMetric, we use $\epsilon = 0.2$, $w = 200$, and the maximum number of rewriting steps is $n = 500$.

Summary of results. The results of this evaluation are shown in Fig. 9(a). SyMetric can solve 77% of these benchmarks, but the baselines fail on all of them except at most 3. In what follows, we discuss why the baselines perform poorly and the failure cases for SyMetric.

FTA results. The FTA baseline fails on all but the smallest of the handwritten benchmarks. When it fails, it is universally because it runs out of memory. For this domain, few programs produce exactly the same output image, so equivalence reduction is not enough to reduce memory consumption.

AFTA results. We found that the Syngar baseline cannot solve *any* benchmarks when we include the `Repeat` operator in the DSL, as repetition causes difficulties with Syngar's abstraction refinement phase, specifically because refining one pixel can cause the refinement of multiple other pixels, which causes the abstraction to track an increasing number of pixels over multiple refinement iterations [Wang et al. 2018]. However, Syngar can solve 3 of the 47 benchmarks if we omit the `Repeat` operator from the DSL. For many of the remaining benchmarks, the target programs become quite complex without the `Repeat` operator, so Syngar fails either because it reaches the time limit or fails to find a program with the AST size limit of 40.

There are two key differences that make the CSG domain more tractable with our approach than with Syngar. First, our algorithm uses the distance function for guidance during the search. Syngar is directed only by the abstraction, which needs to separate useful subprograms from useless ones. This is a strong requirement, and we found that it was difficult to construct an abstraction that could perform this function. Second, because Syngar does not have a local search, if the abstraction ever mixes the correct program with some incorrect programs, it will need to refine before it can succeed.

Sketch-Du results. Our third baseline Sketch-Du—an adaptation of InverseCSG [Du et al. 2018] to our setting—can solve three of the handwritten benchmarks and one of the generated benchmarks but fails on the remaining ones. For the 43 benchmarks it cannot solve, it runs out of memory 77% of the time and runs out of time the remaining 15%. We attempted to provide this baseline with parameters that would minimize its memory use and maximize its chances of successfully completing the benchmarks. To that end, we used Sketch's specialized integer solver to reduce memory overhead; we controlled the amount of unrolling in the sketch based on the size of the benchmark program, and we used Sketch's example file feature, which reduces the time to find counterexamples during the CEGIS loop. However, even with these optimizations, we found that the large number of examples in the inverse CSG domain (one per pixel, so 1024 total) causes Sketch to perform many iterations of the CEGIS loop.

SyMetric results. SyMetric performs significantly better than the other baselines, solving 77% of the benchmarks. Based on our manual inspection, we found two dominant failure modes. One of them is that the beam width w may be too narrow, causing critical subprograms to be dropped from the search space. This effect is more pronounced when the benchmark relies on subprograms that are far from the goal O according to δ. One pattern that we noticed among the failure cases is that they include subprograms where one shape is subtracted from another, producing a complex shape that is distant from its inputs and also distant from the final image. The second way that a benchmark can fail is that a program close to the solution is contained in the XFTA, but the Extract and Repair procedures are unable to find it. While extracting all programs accepted by the XFTA could mitigate the problem, the overhead of doing so is often prohibitively expensive.

6.2 Regular Expression Synthesis

Our second application domain is regular expression synthesis. Given a sketch and a set of positive and negative examples, the task is to find a regular expression that conforms to the given sketch and matches all positive examples, while matching none of the negative ones.

For this application domain, we perform an empirical comparison against the following baselines:

- Regel: This baseline is a state-of-the-art regular expression synthesis tool [Chen et al. 2020]. It performs *top-down* (rather than bottom-up) synthesis and uses several SMT-based pruning strategies to reduce the search space.
- FTA: This baseline performs bottom-up enumeration with equivalence reduction using FTAs for our regular expression DSL.
- AFTA: As in the Inverse CSG domain, this baseline is an abstraction-based version of bottom-up enumeration with equivalence reduction [Wang et al. 2018]. We use a predicate abstraction that tracks the length of the longest match and the beginning of the first match. These predicates effectively allow the tool to avoid examining programs which do not match the whole string or that do not match from the beginning.

We note that the FTA and AFTA baselines use the same interpreter as SyMetric. However, Regel uses a different regular expression matching algorithm based on the Brics automaton library, which is not as efficient as our optimized implementation for this DSL.

Benchmarks. For this experiment we use the Stack Overflow dataset taken from [Chen et al. 2020]. This benchmark was collected from user questions. It contains 122 distinct tasks, which consist of a natural language description and a set of input-output examples. For each task, Chen et al. automatically generates a set of *sketches* (i.e. partial programs) that capture additional constraints about the target regular expression that are present in the natural language description. This gives us a total of 2173 total task/sketch pairs to use in our evaluation. However, we note that some of these synthesis problems may not be solvable since the generated sketches could be wrong. As discussed in §4.2, we treat the sketches as additional DSL operators, so they are used by all the algorithms.

Experimental setup. These experiments are run on the same machine as the CSG benchmarks. We use a time limit of 5 minutes. We run in a "high memory" mode that has a 4GB limit and a "low memory" mode that has a 100MB limit. We use $\epsilon = 0.3$, $w = 200$, and $n = 100$ for the regular expression domain.

Results summary. The results of this experiment are summarized in Fig. 9(c) (high memory) and Fig. 9(d) (low memory). Among the four tools, SyMetric achieves the best performance, solving 74/71% (high/low%) of the regex benchmarks, compared to Regel's 59/48%. The FTA baseline significantly outperforms the abstraction-based approach, solving 72/51% (for FTA) compared to 13/13% (for AFTA).

We chose to present results for two different memory limits to show that SyMetric is significantly more memory efficient than FTA. While FTA is competitive with SyMetric when given 4GB of memory, it is unable to solve the harder benchmarks when given less, because it runs out of room to store the increasingly large state space. In contrast, SyMetric achieves similar performance in both settings because it retains many fewer states.

One caveat about these results is that, while the comparison between SyMetric, FTA, and AFTA is apples-to-apples, Regel uses a different (and less efficient) interpreter for their DSL.[2] Another caveat is that, while our predicate abstraction does not yield good results, it *may* be possible to build abstractions that perform better in the regex domain. Nevertheless, we believe these results substantiate our claim that (a) SyMetric is competitive with state-of-the-art tools for the regex domain, and (b) metric program synthesis yields improvements over basic observational equivalence reduction.

6.3 Solving Tower Building Tasks

As our third application domain, we consider tower building tasks that are inspired by planning in AI and that were used for evaluating program synthesis tools in prior work [Ellis et al. 2021; Nye et al. 2021]. As explained in §4, the goal of this task is to generate a program that constructs a given configuration of blocks.

For this domain, we compare SyMetric against two baselines:

- Neural: Our first baseline is a state-of-the-art neural synthesizer [Nye et al. 2021] that combines top-down program synthesis with a neural network that predicts the possible outputs of a partial program. Since this baseline is trained on a set of representative tower building tasks, it can use tower motifs encountered during training to solve new tasks.

[2]We believe it is due to this implementation difference in the interpreter that FTA slightly outperforms Regel. The pruning heuristics used in Regel allow it to scale without needing a custom interpreter.

- FTA: As in the previous two domains, our second baseline performs bottom-up enumerative search with equivalence reduction using FTAs.

For this domain, we do not compare SyMetric against the abstraction-based FTA approach, as the combination of loops and mutable state make this domain difficult grounds for applying prior work [Wang et al. 2018]. In fact, we believe that applying abstraction refinement techniques to this domain/DSL is an open research problem in its own right.

Benchmarks. Our tower building benchmarks are drawn from [Nye et al. 2021], which are constructed by systematically composing tower building programs together (e.g., taking a program that builds a $w \times h$ bridge and building two side-by-side, varying the size and spacing). The original benchmark suite contains 40 tasks, but we found that two of the tasks are duplicates and four are not expressible in the DSL described in [Nye et al. 2021], as they require loops with non-constant iteration counts. We remove these six tasks, resulting in a total of 34 tower building benchmarks used in our evaluation.

Experimental setup. These experiments are run on the same machine as the CSG benchmarks. We use a time limit of 10 minutes and a memory limit of 4 GB. For hyperparameters, we use $\epsilon = 0.4$, $w = 100$, and $n = 100$.

Results summary. The results for this domain are summarized in Fig. 9(b). Note that we do not show running time for Neural, as it is not reported in [Nye et al. 2021] and we do not have access to their model. Overall, we find that SyMetric approaches the performance of the neural approach and that it performs significantly better than FTA. In particular, FTA performs poorly in this domain because the target programs tend to be fairly large and few of the enumerated programs result in the same block configuration, so equivalence reduction is not as effective in this context. SyMetric deals with the large search space size by exploiting observational similarity and by using ranking to select promising subprograms. Finally, we note that, while Neural can solve a few more benchmarks compared to SyMetric, it can do so by using motifs learned from training data as building blocks. In contrast, SyMetric can achieve similar performance without requiring access to training data.

6.4 Detailed Evaluation of Metric Synthesis

To gain more insights about the effectiveness of metric program synthesis and answer RQ2 and RQ3, we perform a detailed evaluation of SyMetric in the inverse CSG domain. We explore distance-based clustering in more depth and present the results of relevant ablation studies.

6.4.1 Ablation Studies. In this section, we describe a set of ablation studies to evaluate the relative importance of the algorithms used in our approach. We consider the following ablations:

- NoCluster: This variant does not perform clustering during FTA construction. However, it still performs repair after extracting a program from the FTA.
- NoRank: This variant does not use distance-based ranking during XFTA construction. Instead, it picks w randomly chosen (clustered) states to add to the automaton in each iteration.
- ExtractRandom: This variant does not use our proposed distance-based program extraction technique. Instead, it randomly picks programs that are accepted by the automaton. (However, the order in which final states are considered is still determined using the distance metric.)
- RepairRandom: This variant does not use our distance-based program repair technique. Instead, after applying a rewrite rule during the Repair procedure, it randomly picks one of the programs rather than using the distance metric to pick the one closest to the goal.
- SimpleDistance: This variant uses the Jaccard distance instead of the distance proposed in §4.1.3.

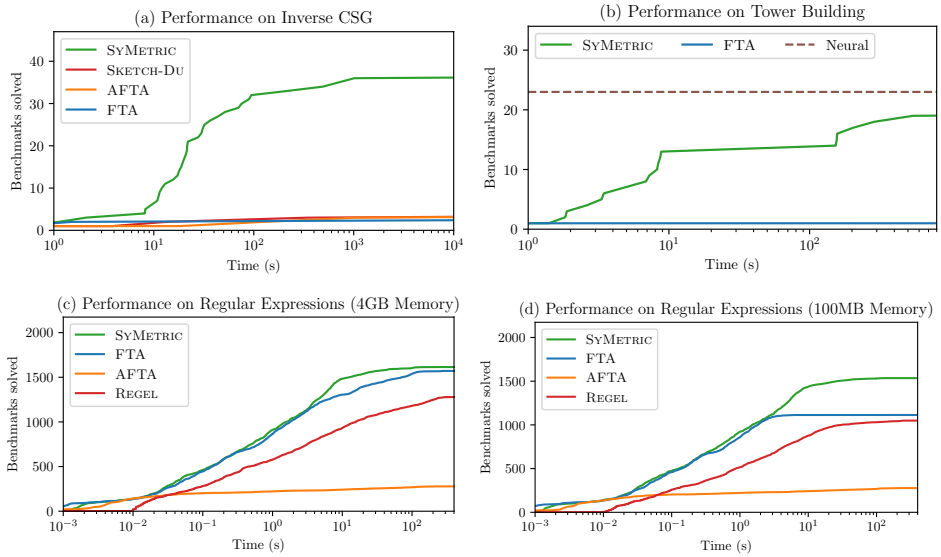

Fig. 9. Synthesis performance on the inverse CSG, tower building, and regular expression domains.

Benchmark	ConstructXFTA	Extract	Repair	Expansion	Clustering	Ranking
Generated	19.7/30.6	3.1/599.9	1.7/396.1	11.3/18.4	2.4/4.6	0.0/0.1
Hand-written	11.1/27.3	0.9/43.5	1.1/42.9	5.5/12.7	2.4/8.3	0.0/0.2
All	16.4/30.6	1.2/599.9	1.2/396.1	10.3/18.4	2.4/8.3	0.0/0.2

Fig. 10. Runtime breakdown (median/max, in seconds) for sub-procedures of SyMetric and ConstructXFTA on inverse CSG.

The results of these ablation studies are presented in Fig. 11 (again, for the inverse CSG domain). For the ablations, we use the randomly generated CSG benchmarks, because they have more uniform difficulty. We find that, for this domain, the most important component of our algorithm is the distance-guided Repair procedure, followed by ranking during XFTA construction, and then clustering. The distance-guided Extract procedure has less impact, but fewer programs are synthesized if we randomly choose a program instead of using the distance metric for extraction. Finally, the distance function has a significant impact on the speed of synthesis and the number of programs solved. This supports our discussion in §4.1.3 of the importance of local reachability.

We have observed in some cases that disabling ranking yields performance improvements for easy benchmarks. While ranking is cheap on its own, if it is disabled, we only need to enumerate new states until we can build w clusters. When ranking is enabled, we need to look at the entire frontier at least once to sort it. However, our ablation shows that ranking has a huge positive impact for the harder benchmarks and without it, the number of benchmarks solved drops significantly.

We do not perform an ablation that considers clustering on its own; that is, without limiting the number of clusters that are collected. This is because clustering is $O(n^2)$ (Algorithm 3), so it is too expensive to cluster the entire frontier without increasing the cluster size excessively.

6.4.2 Detailed Evaluation of Running Time. In this section, we explore the impact of different sub-procedures on running time, again on the inverse CSG domain. Figure 10 compares the running

times of XFTA construction (ConstructXFTA), program extraction (Extract), program repair (Repair), and the expansion, clustering and ranking subprocedures of ConstructXFTA.

ConstructXFTA usually dominates total synthesis time. In contrast, program extraction from the XFTA using our greedy approach is quite fast, taking a median of 1.2 seconds. Finally, while the median running time of Repair is 1.2 seconds, it varies widely depending on how many calls to Repair are made and how many rewrite rules we need to apply to find the correct program. Regarding the subprocedures of ConstructXFTA, the expansion phase dominates XFTA construction time. This is not surprising because expansion requires evaluating DSL programs to construct new states. Ranking barely takes any time and clustering takes a median of 2.4 seconds.

6.4.3 Evaluation of Benchmark Scaling. To illustrate the effect of benchmark size on SyMetric's performance, we generate 20 random benchmarks of sizes 20 to 100. Ideally, we would generate *minimal* programs of a given size, but this is difficult. We cannot cheaply check that the programs we generate are minimal, but we check that every subprogram in the generated program cannot be removed without affecting the program's behavior. If a removing a subprogram does not affect the program's behavior, then it must be non-minimal. This is a more aggressive check than we used for the random programs discussed in §6.1.

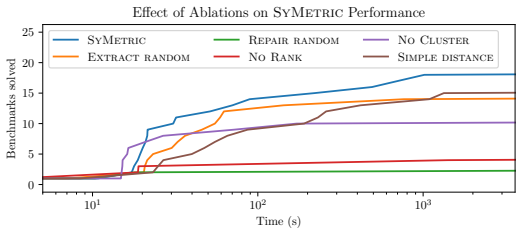

Fig. 11. Effect of ablations on SyMetric for inverse CSG.

We run SyMetric with the same hyperparameters, time, and memory limits as in §6.1. Results are shown in Fig. 12. These results show that SyMetric performs well on programs with up to 20 AST nodes, but its performance drops quickly as the programs grow in size. This indicates that some form of partitioning would be necessary to achieve reasonable performance on complex CSG models.

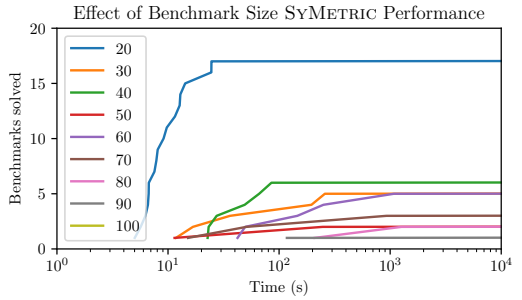

Fig. 12. Relationship between program size and SyMetric synthesis performance.

6.5 Large Language Model Case Study

Given the recent explosion of interest in applying LLMs to program synthesis, we also perform a case study applying LLMs to our evaluation domains. For each benchmark, we construct a prompt that includes a natural language description of the DSL syntax and semantics and two examples of solved synthesis problems, consisting of program output followed by the correct program. The prompt is provided to the model and the model is queried for ten completions. We treat the problem as solved if any of the completions is correct. We run a systematic evaluation using the GPT-3.5 [OpenAI 2022] API and we manually run several prompts through the GPT-4 [OpenAI 2023] web interface. We do not observe a difference in performance between GPT-3.5 and GPT-4 from these manual inspections. We find that the model frequently generates syntactically well-formed programs (which is impressive considering it is presented with the DSL in the prompt) but it rarely generates correct programs.

We obtain the following results with GPT-3.5. We report the best result of the ten completions.

- **Inverse CSG:** On the 40 tasks, the model generates 0 correct and 40 well-formed programs.
- **Regular Expressions:** On the 2173 task/sketch pairs, the model generates 111 correct programs, 34 programs that match the examples but not the sketch, 1135 programs that match the sketch but not the examples, and 255 well-formed programs. The remainder are not well-formed. Many of the sketches simply assert that the program contains some subterm; the model can generate a matching program by simply repeating part of the sketch.
- **Tower Building:** On the 34 tasks, the model generates 0 correct and 34 well-formed programs.

While these results are poor, LLM performance is sensitive to the prompting strategy [Si et al. 2023], so there may be a better prompting method that we did not discover. Multi-modal LLMs [Alayrac et al. 2022] may perform better on our image-based tasks. Nonetheless, we believe these results further demonstrate that the tasks in our evaluation are quite hard.

7 RELATED WORK

Bottom-up Synthesis. Our work builds heavily on bottom-up synthesis with equivalence reduction, an idea that was introduced concurrently by Albarghouthi et al. and Udupa et al.. Later work by Wang et al. explored another variation of this idea in the context of version space learning and showed how to use Finite Tree Automata (FTA) to compactly represent the space of programs consistent with a given specification. Our work was particularly inspired by Blaze [Wang et al. 2018] which uses abstraction refinement to speed up bottom-up search. Abstractions provide a mechanism for grouping closely related solutions, allowing large sets of solutions to be ruled out by evaluating only one abstract solution. In this regard, Blaze can be viewed as performing equivalence reduction over abstract domains. In this work, we explore another relaxation of observational equivalence based on the notion of observational similarity rather than abstract equivalence. We believe that our metric program synthesis idea is complementary to the abstraction refinement approach and can work well in settings for which abstract domains are hard to design or where abstractions do not effectively reduce the search space.

Quantitative Synthesis. Our algorithm uses distance metrics to group similar programs and to rank them. In this regard, our method bears similarities to prior work exploring quantitative goals in program synthesis, both in the reactive synthesis space [Cerný and Henzinger 2011] and in functional synthesis (e.g. [Schkufza et al. 2013, 2014]). In many of these cases, the quantitative objectives are used to deal with noisy or probabilistic specifications [Handa and Rinard 2020; Raychev et al. 2016], where we use them to perform search more effectively. For example, Raychev et al. combine a synthesizer with a dataset sampler in a way that reduces the likelihood of overfitting to an erroneous point in the dataset. In contrast, we assume that the examples provided are correct; error is introduced by our approximation of the search space.

We use local search to recover from the approximation introduced by the XFTA. Santolucito et al. apply gradient descent to synthesize digital signal processing programs in a way that is similar to our use of local search. A key difference is that in our domains, the solutions have nontrivial "discrete parts" (i.e., the structure of the correct program, modulo the parts modifiable by local search). This motivates our use of XFTAs, as a way to search over the discrete portion of the program.

Neural-guided Synthesis. There has been significant interest in neural-guided program synthesis, where a neural network is trained to guide the search for a program that satisfies a specification. In early incarnations of this idea, the neural network was used simply to select components that were likely to be used by the program [Balog et al. 2017], but starting with the work of Devlin et al., the neural network has been heavily involved in directing the search. Especially relevant

to our work is the work on execution guided synthesis [Chen et al. 2019; Ellis et al. 2019], which uses the state of the partially constructed program to determine the most promising next step for the synthesizer. The work by Ellis et al. in particular inspired the ranking phase of our current algorithm. That work uses a learned value function—trained to evaluate the output of candidate programs—to determine which to keep as part of the beam. A common limitation of all the neural guided synthesis approaches is that they require significant work ahead of time to collect a dataset and train the algorithm on that dataset. In contrast, our approach relies on a domain-specific distance function, and we find that simple distance functions often work fairly well. Potential future work could apply the insights of this work in a deep learning context.

Genetic Programming. The genetic programming community has explored the use of distance functions between observed program behaviors [Moraglio et al. 2012], clustering [Collet et al. 2006], and diversity [Burke et al. 2004]. Lexicase selection [Helmuth et al. 2014; La Cava et al. 2016] is an interesting alternative to the ranking step of our algorithm. It improves population diversity by selecting programs that perform particularly well on small subsets of the test cases.

Despite these shared ideas, our algorithm has several key differences from existing work in genetic programming. First, the focus on enumeration is a key part of our algorithm. It induces a strong bias towards short programs, which has always been a focus in the synthesis field and improves the generalization properties of synthesized programs. Second, our algorithm constructs a large, compact program space, represented as an XFTA; genetic approaches work with sets of individual programs, which increases the cost of retaining a large space. The large space is valuable during the program extraction phase. Finally, our approach is a generalization of an existing, widely used synthesis algorithm—bottom-up synthesis with equivalence reduction.

Diversity. One effect of the grouping performed by our algorithm during search is to increase the diversity of the programs in the beam, allowing it to cover more distinct programs and increasing the chance a program close to the goal will be included. There has been some prior work on using diversity measures as part of search. The genetic programming community has long recognized that diversity in a population of programs is crucial to avoid converging to low-quality local minima and has explored many diversity measures to maintain diversity of the population [Burke et al. 2004]. In the context of beam search for NLP, there has been recent recognition that the top-K elements of a beam may be too close to each other and fail to capture multiple modes in the underlying distribution, which has led to the proposal of *Diverse Beam Search* to force proposals in a beam to be sufficiently different from each other [Vijayakumar et al. 2018]. Our work shares some intuitions with some of these prior works, but to our knowledge, this paper is the first to apply the idea of grouping based on a similarity function in the context of FTA-based synthesis.

Program Synthesis for Inverse CSG. There has been a lot of interest in the CAD community in using program synthesis techniques to reverse engineer CAD problems. Two early works in this space are InverseCSG [Du et al. 2018] and the work of Nandi et al.. Both aim to reverse engineer 3D CSG programs from meshes, but both rely on specialized algorithms to do much of the work. Nandi et al. rely on a set of domain specific oracles that examine the mesh and generate proposed decompositions (i.e. splitting the mesh into a union of two simpler shapes). These oracles are powerful, but highly specialized to the CSG domain. Similarly, InverseCSG uses a preprocessing phase to identify all constituent primitive shapes and their parameters, so the synthesizer only has to discover the Boolean structure of the shape. In contrast, our synthesis method can solve for all primitives and their parameters without relying on a domain-specific preprocessor. InverseCSG also relies on a segmentation algorithm to break large shapes into small fragments that are then assembled into the final shape. In contrast, we aim to solve the entire inverse CSG problem as

a single synthesis task. Finally, our program space is richer than that of InverseCSG because it includes a looping construct as well as the primitives and Boolean operations. Our experimental results show that we can do with a single algorithm what in prior work required an entire pipeline of complex and specialized algorithms.

Another line of work uses neural networks to recover structured CAD models from unstructured input [Sharma et al. 2018; Tian et al. 2019]. Their performance is generally excellent, but dependent on training data (e.g., Shape2Prog [Tian et al. 2019] is specialized to just furniture shapes). One of the goals of our algorithm is to only require limited domain knowledge, expressed in the distance function and repair rules. However, we believe that our approach can complement the neural methods, either by using a learned distance metric or by using a network to predict a likely space of programs, as in [Lee et al. 2018].

Further prior work processes CSG programs, either to extract common structure [Jones et al. 2021] or to capture regularity [Nandi et al. 2020]. ShapeMOD [Jones et al. 2021], for example, extracts common macros from a library of CSG programs. These macros form a domain specific language for a class of CSG programs (e.g., furniture) and help provide physically plausible results that are biased towards patterns common in the target domain. Our method does not rely on a set of training programs, but could use macros like these if they were available. Nandi et al. show that it is possible to post-process the output of an InverseCSG-like system to extract loops, which produces programs that are more general and easier to modify. However, this approach requires first synthesizing the loop free program, which can be large for models with a lot of repetition.

Synthesis of Regular Expressions. Synthesis of regular expressions from examples has a long history. Recent work includes Regel [Chen et al. 2020] and AlphaRegex [Lee et al. 2016], which are top-down synthesizers that use upper and lower bounds for pruning. Regel also uses natural language descriptions to improve generalizability. It parses the natural language descriptions into so-called *hierarchical sketches* and uses top-down enumerative search combined with SMT-based pruning to find a sketch completion that satisfies the examples. In our evaluation, we use the sketches generated by Regel but solve them using metric program synthesis instead of top-down SMT-guided search. Repair of regular expressions, which is related to our local search, has also attracted significant attention from the research community [Pan et al. 2019; Rebele et al. 2018]. Because our method starts the local search process with programs that are already fairly close to the goal, we can use simpler rewrite-based techniques for repair.

Synthesis of Tower Building Programs. The tower building domain first appears in [Ellis et al. 2021] and is used as a benchmark in [Nye et al. 2021]. The appeal of this domain comes from its loops and mutable state, as well as its connection to classical AI planning tasks. Prior work has focused on the application of neural guided synthesis to this domain. We show that a non-neural approach can also perform well.

8 CONCLUSION

We presented a new synthesis method, called *metric program synthesis*, that performs search space reduction using a distance metric. The key idea behind our technique is to cluster similar states during bottom-up enumeration and then perform program repair once a program that is "close enough" to the goal is found. Our approach constructs a so-called *approximate finite tree automaton* that represents a set of programs that "approximately" satisfy the specification. Our method then repeatedly extracts programs from this set and uses distance-guided rewriting to find a repair that exactly satisfies the given input-output examples.

We formalize our intuitions about which DSLs work for our proposed algorithm and show that under these (strong) conditions, our algorithms is complete. We use these conditions as a guide to

instantiate our synthesis framework in three different domains (inverse CSG, regular expression synthesis, and tower building) by defining suitable distance metrics. Our evaluation shows that our tool, SyMetric, outperforms prior domain-agnostic FTA-based techniques in these domains. Furthermore, we compare our approach against domain-specific synthesizers and show that the performance of SyMetric is competitive with these tools despite not using any training data or domain-specific synthesis algorithms.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1918889. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, Roman Ring, Eliza Rutherford, Serkan Cabi, Tengda Han, Zhitao Gong, Sina Samangooei, Marianne Monteiro, Jacob L Menick, Sebastian Borgeaud, Andy Brock, Aida Nematzadeh, Sahand Sharifzadeh, Mikoł aj Bińkowski, Ricardo Barreira, Oriol Vinyals, Andrew Zisserman, and Karén Simonyan. 2022. Flamingo: a Visual Language Model for Few-Shot Learning. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 23716–23736. https://proceedings.neurips.cc/paper_files/paper/2022/file/960a172bc7fbf0177ccccbb411a7d800-Paper-Conference.pdf

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. 934–950. https://doi.org/10.1007/978-3-642-39799-8_67

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. https://openreview.net/forum?id=ByldLrqlx

E.K. Burke, S. Gustafson, and G. Kendall. 2004. Diversity in genetic programming: an analysis of measures and correlation with fitness. *IEEE Transactions on Evolutionary Computation* 8, 1 (2004), 47–62. https://doi.org/10.1109/TEVC.2003.819263

Pavol Cerný and Thomas A. Henzinger. 2011. From boolean to quantitative synthesis. In *Proceedings of the 11th International Conference on Embedded Software, EMSOFT 2011, part of the Seventh Embedded Systems Week, ESWeek 2011, Taipei, Taiwan, October 9-14, 2011*, Samarjit Chakraborty, Ahmed Jerraya, Sanjoy K. Baruah, and Sebastian Fischmeister (Eds.). ACM, 149–154. https://doi.org/10.1145/2038642.2038666

Qiaochu Chen, Aaron Lamoreaux, Xinyu Wang, Greg Durrett, Osbert Bastani, and Isil Dillig. 2021. Web question answering with neurosymbolic program synthesis. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. 328–343. https://doi.org/10.1145/3453483.3454047

Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-modal synthesis of regular expressions. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. 487–502. https://doi.org/10.1145/3385412.3385988

Xinyun Chen, Chang Liu, and Dawn Song. 2019. Execution-Guided Neural Program Synthesis. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=H1gfOiAqYm

Paolo Ciaccia, Marco Patella, and Pavel Zezula. 1997. M-tree: An efficient access method for similarity search in metric spaces. In *Vldb*, Vol. 97. 426–435.

Pierre Collet, Marco Tomassini, Marc Ebner, Steven M. Gustafson, and Anikó Ekárt (Eds.). 2006. *Genetic Programming, 9th European Conference, EuroGP 2006, Budapest, Hungary, April 10-12, 2006, Proceedings*. Lecture Notes in Computer Science, Vol. 3905. Springer. https://doi.org/10.1007/11729976

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, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. 990–998. http://proceedings.mlr.press/v70/devlin17a.html

Tao Du, Jeevana Priya Inala, Yewen Pu, Andrew Spielberg, Adriana Schulz, Daniela Rus, Armando Solar-Lezama, and Wojciech Matusik. 2018. Inversecsg: Automatic Conversion of 3d Models To CSG Trees. *ACM Trans. Graph.* 37, 6 (2018), 213:1–213:16. https://doi.org/10.1145/3272127.3275006

Kevin Ellis, Maxwell I. Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. 2019. Write, Execute, Assess: Program Synthesis with a REPL. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 9165–9174. https://proceedings.neurips.cc/paper/2019/hash/50d2d2262762648589b1943078712aa6-Abstract.html

Kevin Ellis, Catherine Wong, Maxwell I. Nye, Mathias Sablé-Meyer, Lucas Morales, Luke B. Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2021. DreamCoder: bootstrapping inductive program synthesis with wake-sleep library learning. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. 835–850. https://doi.org/10.1145/3453483.3454080

Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. 420–435. https://doi.org/10.1145/3192366.3192382

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 *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 422–436. https://doi.org/10.1145/3062341.3062351

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 229–239. https://doi.org/10.1145/2737924.2737977

Fred Glover and Manuel Laguna. 1998. Tabu Search. In *Handbook of Combinatorial Optimization: Volume1–3*, Ding-Zhu Du and Panos M. Pardalos (Eds.). Springer US, 2093–2229. https://doi.org/10.1007/978-1-4613-0303-9_33

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. 317–330. https://doi.org/10.1145/1926385.1926423

Shivam Handa and Martin C. Rinard. 2020. Inductive program synthesis over noisy data. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. 87–98. https://doi.org/10.1145/3368089.3409732

Thomas Helmuth, Lee Spector, and James Matheson. 2014. Solving uncompromising problems with lexicase selection. *IEEE Transactions on Evolutionary Computation* 19, 5 (2014), 630–643.

R. Kenny Jones, David Charatan, Paul Guerrero, Niloy J. Mitra, and Daniel Ritchie. 2021. Shapemod: Macro Operation Discovery for 3d Shape Programs. *ACM Trans. Graph.* 40, 4 (2021), 153:1–153:16. https://doi.org/10.1145/3450626.3459821

William La Cava, Lee Spector, and Kourosh Danai. 2016. Epsilon-lexicase selection for regression. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. 741–748.

Tessa A. Lau, Steven A. Wolfman, Pedro M. Domingos, and Daniel S. Weld. 2003. Programming by Demonstration Using Version Space Algebra. *Mach. Learn.* 53, 1-2 (2003), 111–156. https://doi.org/10.1023/A:1025671410623

Mina Lee, Sunbeom So, and Hakjoo Oh. 2016. Synthesizing regular expressions from examples for introductory automata assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2016, Amsterdam, The Netherlands, October 31 - November 1, 2016*. 70–80. https://doi.org/10.1145/2993236.2993244

Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating search-based program synthesis using learned probabilistic models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. 436–449. https://doi.org/10.1145/3192366.3192410

Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. 2022. Bottom-up synthesis of recursive functional programs using angelic execution. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–29. https://doi.org/10.1145/3498682

Alberto Moraglio, Krzysztof Krawiec, and Colin G Johnson. 2012. Geometric semantic genetic programming. In *Parallel Problem Solving from Nature-PPSN XII: 12th International Conference, Taormina, Italy, September 1-5, 2012, Proceedings, Part I 12*. Springer, 21–31.

Chandrakana Nandi, James R. Wilcox, Pavel Panchekha, Taylor Blau, Dan Grossman, and Zachary Tatlock. 2018. Functional programming for compiling and decompiling computer-aided design. *Proc. ACM Program. Lang.* 2, ICFP (2018), 99:1–99:31. https://doi.org/10.1145/3236794

Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing structured CAD models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 31–44. https://doi.org/10.1145/3385412.3386012

Maxwell I. Nye, Yewen Pu, Matthew Bowers, Jacob Andreas, Joshua B. Tenenbaum, and Armando Solar-Lezama. 2021. Representing Partial Programs with Blended Abstract Semantics. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. https://openreview.net/forum?id=mCtadqIxOJ

OpenAI. 2022. *Introducing ChatGPT*. https://openai.com/blog/chatgpt

OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

Rong Pan, Qinheping Hu, Gaowei Xu, and Loris D'Antoni. 2019. Automatic Repair of Regular Expressions. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 139:1–139:29. https://doi.org/10.1145/3360565

Matt Pharr and William R Mark. 2012. ispc: A SPMD compiler for high-performance CPU programming. In *2012 Innovative Parallel Computing (InPar)*. IEEE, 1–13.

Veselin Raychev, Pavol Bielik, Martin T. Vechev, and Andreas Krause. 2016. Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 761–774. https://doi.org/10.1145/2837614.2837671

Thomas Rebele, Katerina Tzompanaki, and Fabian M. Suchanek. 2018. Adding Missing Words to Regular Expressions. In *Advances in Knowledge Discovery and Data Mining - 22nd Pacific-Asia Conference, PAKDD 2018, Melbourne, VIC, Australia, June 3-6, 2018, Proceedings, Part II*. 67–79. https://doi.org/10.1007/978-3-319-93037-4_6

Mark Santolucito, Kate Rogers, Aedan Lombardo, and Ruzica Piskac. 2018. Programming-by-example for audio: synthesizing digital signal processing programs. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design, FARM@ICFP 2018, St. Louis, MO, USA, September 29, 2018*, Brent Yorgey and Donya Quick

(Eds.). ACM, 18–25. https://doi.org/10.1145/3242903.3242906

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*. 305–316. https://doi.org/10.1145/2451116.2451150

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic optimization of floating-point programs with tunable precision. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 53–64. https://doi.org/10.1145/2594291.2594302

Vadim Shapiro and Donald L. Vossler. 1991. Construction and optimization of CSG representations. *Comput. Aided Des.* 23, 1 (1991), 4–20. https://doi.org/10.1016/0010-4485(91)90077-A

Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhransu Maji. 2018. CSGNet: Neural Shape Parser for Constructive Solid Geometry. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. 5515–5523. https://doi.org/10.1109/CVPR.2018.00578

Chenglei Si, Zhe Gan, Zhengyuan Yang, Shuohang Wang, Jianfeng Wang, Jordan Lee Boyd-Graber, and Lijuan Wang. 2023. Prompting GPT-3 To Be Reliable. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=98p5x51L5af

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In *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*. 404–415. https://doi.org/10.1145/1168857.1168907

Yonglong Tian, Andrew Luo, Xingyuan Sun, Kevin Ellis, William T. Freeman, Joshua B. Tenenbaum, and Jiajun Wu. 2019. Learning to Infer and Execute 3D Shape Programs. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. https://openreview.net/forum?id=rylNH20qFQ

Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. 2013. TRANSIT: Specifying Protocols with Concolic Snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. ACM, New York, NY, USA, 287–296. https://doi.org/10.1145/2491956.2462174

Ashwin K. Vijayakumar, Michael Cogswell, Ramprasaath R. Selvaraju, Qing Sun, Stefan Lee, David J. Crandall, and Dhruv Batra. 2018. Diverse Beam Search for Improved Description of Complex Scenes. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*. 7371–7379. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17329

Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2017a. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 452–466. https://doi.org/10.1145/3062341.3062365

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017b. Synthesis of data completion scripts using finite tree automata. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–26.

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2018. Program Synthesis Using Abstraction Refinement. *Proc. ACM Program. Lang.* 2, POPL (2018), 63:1–63:30. https://doi.org/10.1145/3158151

Karl D. D. Willis, Yewen Pu, Jieliang Luo, Hang Chu, Tao Du, Joseph G. Lambourne, Armando Solar-Lezama, and Wojciech Matusik. 2021. Fusion 360 gallery: a dataset and environment for programmatic CAD construction from human design sequences. *ACM Trans. Graph.* 40, 4 (2021), 54:1–54:24. https://doi.org/10.1145/3450626.3459818