Augmented Example-based Synthesis

Semantic Bias in Example-based Synthesis via Relational Perturbation Properties

Shengwei An
Purdue University
an93@purdue.edu

Sasa Misailovic
UIUC
misailo@illinois.edu

Roopsha Samanta
Purdue University
roopsha@purdue.edu

Rishabh Singh
Google Brain
rising@google.com

Abstract

Example-based specifications for program synthesis are inherently ambiguous and may cause synthesizers to generate programs that do not exhibit intended behavior on unseen inputs. Existing synthesis techniques attempt to address this problem by either placing a domain-specific, syntactic bias on the hypothesis space or heavily relying on user feedback to help resolve ambiguity. We present a new framework to address the ambiguity/generalizability problem in example-based synthesis. The key feature of our framework is that it places a semantic bias on the hypothesis space based on *relational perturbation properties*. The framework is portable across multiple domains and synthesizers and is based on two core steps: (1) automatically augment the set of user-provided examples by *applying* relational perturbation properties and (2) use a generic example-based synthesizer to generate a program consistent with the augmented set of examples. Our framework can be instantiated with three different user interfaces, with varying degrees of user engagement to help infer relevant relational perturbation properties. This includes an interface in which the user only provides examples and our framework *automatically infers* relevant properties. We implement our framework in a tool SketchAX specialized to the Sketch synthesizer and demonstrate that SketchAX is effective in significantly boosting the performance of Sketch for all three user interfaces.

Keywords Example-based Synthesis, Max-SMT, Generalizability

1 Introduction

Example-based synthesis, or, Programming By Examples (PBE) [8, 9, 14] is an emerging paradigm of program synthesis that has been applied successfully across diverse domains [6, 8, 13, 20, 22, 23]. The task in PBE is to generate a program from a hypothesis space (often defined as a domain-specific language or DSL) that satisfies a set of input-output (I/O) examples. This example-based specification mechanism can be a double-edged sword. Example-based specifications have made program synthesis more tractable as well as accessible to non-expert users who may not be able to write formal/complete specifications. However, example-based specifications also pose some of the biggest challenges in PBE: *ambiguity-resolution* [7] and the related problem of *generalizability*. Since examples are inherently an ambiguous/incomplete form of specification, there can be a large number of programs that are consistent with a set of examples. Unsurprisingly, not all of these programs may exhibit the (implicit) intended behavior on unseen inputs.

There are two main classes of techniques that have been used to address the ambiguity/generalizability problem in PBE, with some caveats. (1) *Syntactic bias*-based techniques use highly structured DSLs [2, 24] or ranking functions [21] to place a syntactic bias on the hypothesis space. These solutions are either inadequate by themselves or too domain-specific. (2) *User feedback loop*-based techniques employ a user to validate candidate programs or abstract representations of examples, or, answer questions as in active learning [5, 15]. While some of these interaction models [5] are based on principled approaches to address the generalizability problem in PBE, they place a heavy burden on the user that ultimately limits the scope of applicability of PBE.

In this paper, we present a new approach for addressing the ambiguity/generalizability problem in PBE. Our framework is portable across multiple domains and synthesizers, can be instantiated with different user interfaces, and can be used in conjunction with existing techniques based on structured DSLs, ranking functions or user feedback loops. The key feature of our framework is that it places a *semantic bias* on the hypothesis space based on *relational perturbation properties*. While, in general, relational properties may express constraints that relate multiple programs or multiple executions of a single program, relational perturbation properties relate the perturbation/change in a program output to the perturbation/change in a program input. An example of such a property is *permutation invariance*: the program output does not change when the elements of the program input (array) are permuted. Relational perturbation properties enable us to design a simple and efficient solution that is similar, at least in spirit, to *data augmentation* used for improving

PL'18, January 01–03, 2018, New York, NY, USA
2018.

the generalizability of machine learning models [12, 18]. Our core approach is based on two steps: (1) automatically generate an augmented set of examples by *applying* relational perturbation properties to the user-provided examples and (2) use a generic PBE synthesizer to generate a program consistent with the augmented set of examples.

Our solution strategy of enforcing relational properties using examples instead of formal specifications is inspired by two observations: (i) not all PBE synthesizers (e.g. [8]) accept specifications over all inputs and (ii) in cases where a PBE synthesizer accepts such specifications, there is typically a significant performance penalty in terms of synthesis time. We choose relational perturbation properties as they enable us to easily generate additional examples from any set of user-provided examples. For instance, given an I/O example (x, y) consisting of an input array $x = [1, 2, 3]$ and an output $y = 3$, it is trivial to generate additional examples by *applying* permutation invariance: $([3, 2, 1], 3)$, $([2, 1, 3], 3)$, and so on. On the other hand, if we were to use a more general relational property, such as associativity for a program P with two inputs and one output $(\forall x, x', x''.\ P(P(x, x'), x'') = P(x, P(x', x'')))$, the user-provided examples would need to meet several requirements to enable generation of additional examples. For associativity, one would need the user-provided example set to include the examples $((x, x'), y)$, $((y, x''), z)$ and $((x', x''), r)$ in order to generate the single additional example $((x, r), z)$.

So where do the relational perturbation properties come from? Our framework provides three different ways to answer this question using three user interfaces, with varying degrees of user enagement. In User Interface I, the user picks relevant relational perturbation properties in addition to providing examples. In User Interface II, the user provides examples and helps our framework infer relevant properties by validating/invalidating a small set of examples. Finally, in User Interface III, which is identical to the standard PBE setting, the user only provides examples and our framework automatically infers a relevant set of properties using a Partial Max-SMT [3] formulation and a ranking function over property sets. The framework learns the ranking function from a training set comprised of *successful augmented synthesis instances* from User Interface I.

In order to evaluate the efficacy of our technique, we specialize our approach to the Sketch synthesizer [24] and implement it in a tool SketchAX. Our extensive evaluation on a large class of benchmarks demonstrates that SketchAX significantly boosts Sketch's ability to synthesize correct programs for all three user interfaces. For instance, for bit-vector benchmarks that satisfy some relational perturbation properties, SketchAX improves the success rate of Sketch by 157.5%, 157.5% and 155.3%, respectively, for the three user interfaces.

Contributions. Our paper makes the following contributions:

- We present a new approach to address the ambiguity/generalizability problem in PBE. Our approach is based on the novel idea of placing a semantic bias on the hypothesis space using relational perturbation properties (Sec. 4).
- We propose a flexible, portable, synthesizer-agnostic framework that can be instantiated with three different user interfaces, with varying degrees of user engagement to help infer relevant properties (Sec. 5).
- We develop a Partial Max-SMT formulation to automatically infer relevant properties for User Interface III, where the user only provides I/O examples (Sec. 5). We further develop a procedure to learn a ranking function over property sets to drive the Max-SMT formulation (Sec. 6).
- We implement our framework in a tool SketchAX specialized to the Sketch synthesizer (Sec. 6) and demonstrate that SketchAX is effective in significantly boosting the performance of Sketch for all three user interfaces (Sec. 7).

2 Illustrative examples

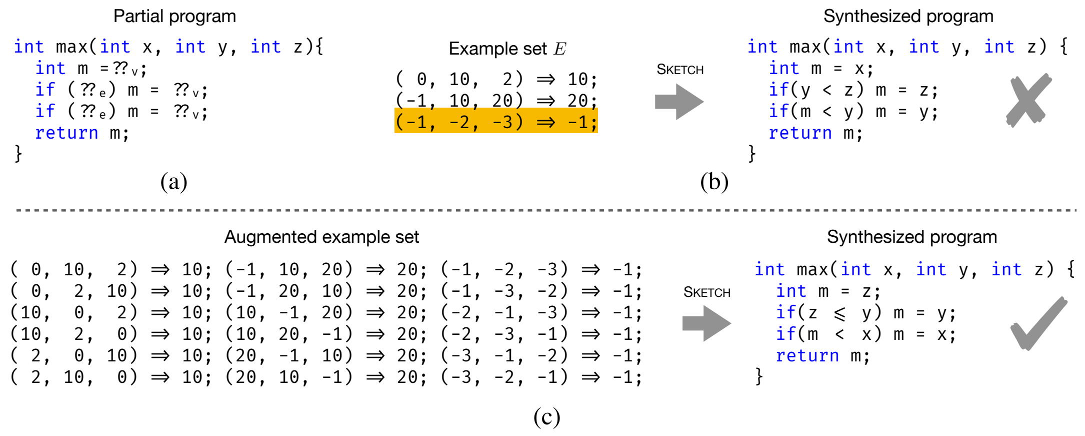
We illustrate the core approach of our framework with a few motivating examples using the Sketch synthesizer.

`max`. Suppose a user wants to synthesize a `max` program that returns the maximum of 3 integer-valued inputs, using Sketch as a PBE synthesizer. A partial program (with holes) that the user may provide is shown in Fig. 1(a). Notice that for the example set E in Fig. 1(b), Sketch yields an incorrect program. Our tool SketchAX addresses this generalizability problem by exploiting the fact that the `max` program should satisfy *permutation-invariance*: the program output should not change if we permute the program inputs. SketchAX automatically augments the initial set of examples E by *applying* the permutation-invariance property to the examples in E as shown in Fig. 1(c). With this augmented set of examples, Sketch is now able to generate the correct `max` program.

Permutation invariance is an instance of a *relational perturbation property* that relates perturbed inputs to corresponding perturbed outputs of programs. Specifically, it is a *structural* perturbation property which changes the relative *positions* of inputs/outputs. The `max` program also satisfies a *value* perturbation property (specifically, *value preservation*) which modifies the values of inputs/outputs. E.g. if we multiply all inputs by some positive constant integer, the output will also be multiplied by the same constant.

We formalize our notion of relational perturbation properties in Sec. 4. Next, we illustrate two useful structural and value perturbation properties.

`matrixTranspose`. The top half of Fig. 2 shows a partial program and an example set E used to synthesize a program to compute the transpose of a matrix. The program generated by Sketch is incorrect. From linear algebra, we know that



```
int max(int x, int y, int z){
  int m =??v;
  if (??e) m = ??v;
  if (??e) m = ??v;
  return m;
}
```

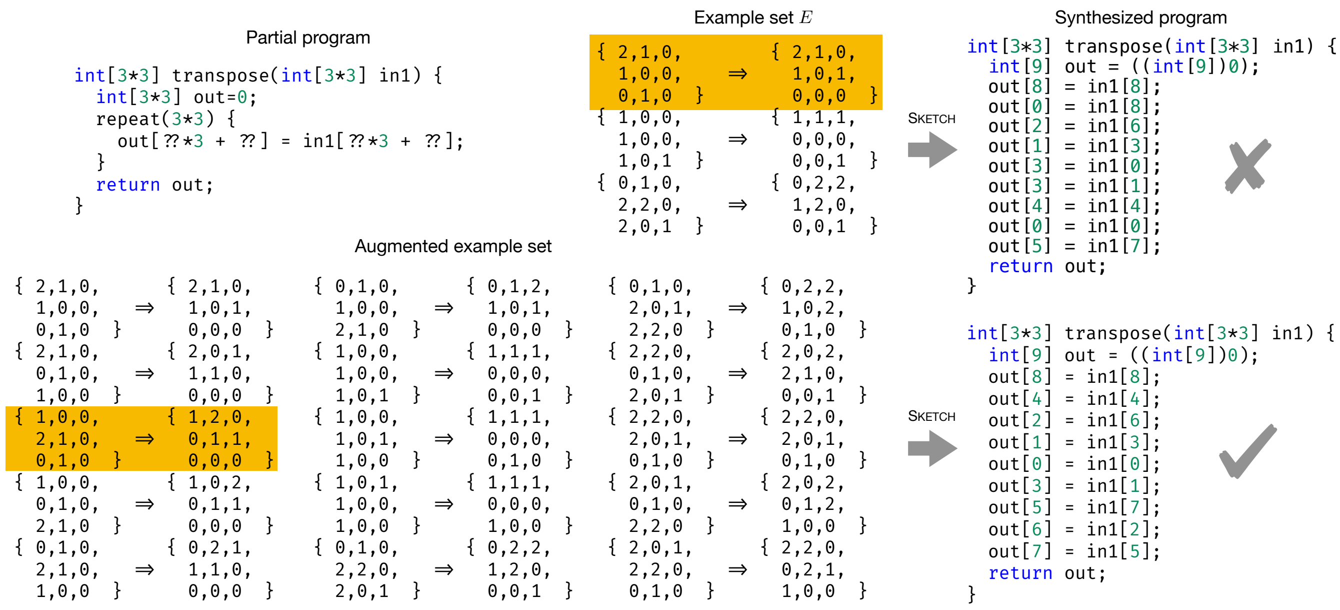
```
( 0, 10,  2) ⇒ 10;
(-1, 10, 20) ⇒ 20;
(-1, -2, -3) ⇒ -1;
```

```
int max(int x, int y, int z) {
  int m = x;
  if(y < z) m = z;
  if(m < y) m = y;
  return m;
}
```

```
( 0, 10,  2) ⇒ 10; (-1, 10, 20) ⇒ 20; (-1, -2, -3) ⇒ -1;
( 0,  2, 10) ⇒ 10; (-1, 20, 10) ⇒ 20; (-1, -3, -2) ⇒ -1;
(10,  0,  2) ⇒ 10; (10, -1, 20) ⇒ 20; (-2, -1, -3) ⇒ -1;
(10,  2,  0) ⇒ 10; (10, 20, -1) ⇒ 20; (-2, -3, -1) ⇒ -1;
( 2,  0, 10) ⇒ 10; (20, -1, 10) ⇒ 20; (-3, -1, -2) ⇒ -1;
( 2, 10,  0) ⇒ 10; (20, 10, -1) ⇒ 20; (-3, -2, -1) ⇒ -1;
```

```
int max(int x, int y, int z) {
  int m = z;
  if(z ⩽ y) m = y;
  if(m < x) m = x;
  return m;
}
```

Figure 1. Computing the maximum of three integers using Sketch and SketchAX.



```
int[3*3] transpose(int[3*3] in1) {
  int[3*3] out=0;
  repeat(3*3) {
    out[??*3 + ??] = in1[??*3 + ??];
  }
  return out;
}
```

```
{ 2,1,0,        { 2,1,0,
  1,0,0,    ⇒     1,0,1,
  0,1,0  }        0,0,0  }
{ 1,0,0,        { 1,1,1,
  1,0,0,    ⇒     0,0,0,
  1,0,1  }        0,0,1  }
{ 0,1,0,        { 0,2,2,
  2,2,0,    ⇒     1,2,0,
  2,0,1  }        0,0,1  }
```

```
{ 2,1,0,      { 2,1,0,      { 0,1,0,      { 0,1,2,      { 0,1,0,      { 0,2,2,
  1,0,0,  ⇒     1,0,1,        1,0,0,  ⇒     1,0,1,        2,0,1,  ⇒     1,0,2,
  0,1,0  }      0,0,0  }      2,1,0  }      0,0,0  }      2,2,0  }      0,1,0  }
{ 2,1,0,      { 2,0,1,      { 1,0,0,      { 1,1,1,      { 2,2,0,      { 2,0,2,
  0,1,0,  ⇒     1,1,0,        1,0,0,  ⇒     0,0,0,        0,1,0,  ⇒     2,1,0,
  1,0,0  }      0,0,0  }      1,0,1  }      0,0,1  }      2,0,1  }      0,0,1  }
{ 1,0,0,      { 1,2,0,      { 1,0,0,      { 1,1,1,      { 2,2,0,      { 2,2,0,
  2,1,0,  ⇒     0,1,1,        1,0,1,  ⇒     0,0,0,        2,0,1,  ⇒     2,0,1,
  0,1,0  }      0,0,0  }      1,0,0  }      0,1,0  }      0,1,0  }      0,1,0  }
{ 1,0,0,      { 1,0,2,      { 1,0,1,      { 1,1,1,      { 2,0,1,      { 2,0,2,
  0,1,0,  ⇒     0,1,1,        1,0,0,  ⇒     0,0,0,        0,1,0,  ⇒     0,1,2,
  2,1,0  }      0,0,0  }      1,0,0  }      1,0,0  }      2,2,0  }      1,0,0  }
{ 0,1,0,      { 0,2,1,      { 0,1,0,      { 0,2,2,      { 2,0,1,      { 2,2,0,
  2,1,0,  ⇒     1,1,0,        2,2,0,  ⇒     1,2,0,        2,2,0,  ⇒     0,2,1,
  1,0,0  }      0,0,0  }      2,0,1  }      0,0,1  }      0,1,0  }      1,0,0  }
```

```
int[3*3] transpose(int[3*3] in1) {
  int[9] out = ((int[9])0);
  out[8] = in1[8];
  out[0] = in1[8];
  out[2] = in1[6];
  out[1] = in1[3];
  out[3] = in1[0];
  out[3] = in1[1];
  out[4] = in1[4];
  out[0] = in1[0];
  out[5] = in1[7];
  return out;
}
```

```
int[3*3] transpose(int[3*3] in1) {
  int[9] out = ((int[9])0);
  out[8] = in1[8];
  out[4] = in1[4];
  out[2] = in1[6];
  out[1] = in1[3];
  out[0] = in1[0];
  out[3] = in1[1];
  out[5] = in1[7];
  out[6] = in1[2];
  out[7] = in1[5];
  return out;
}
```

Figure 2. Synthesizing a function to compute the transpose of a matrix using Sketch and SketchAX.

if we permute the rows of the input matrix, the columns of its transpose will be permuted in the same way. SketchAX applies this perturbation property to E, thereby enabling Sketch to synthesize the correct program. For instance, the highlighted example in E in Fig. 2 is perturbed by swapping the top 2 rows of the input matrix and swapping the left 2 columns of the output matrix to yield the highlighted perturbed example.

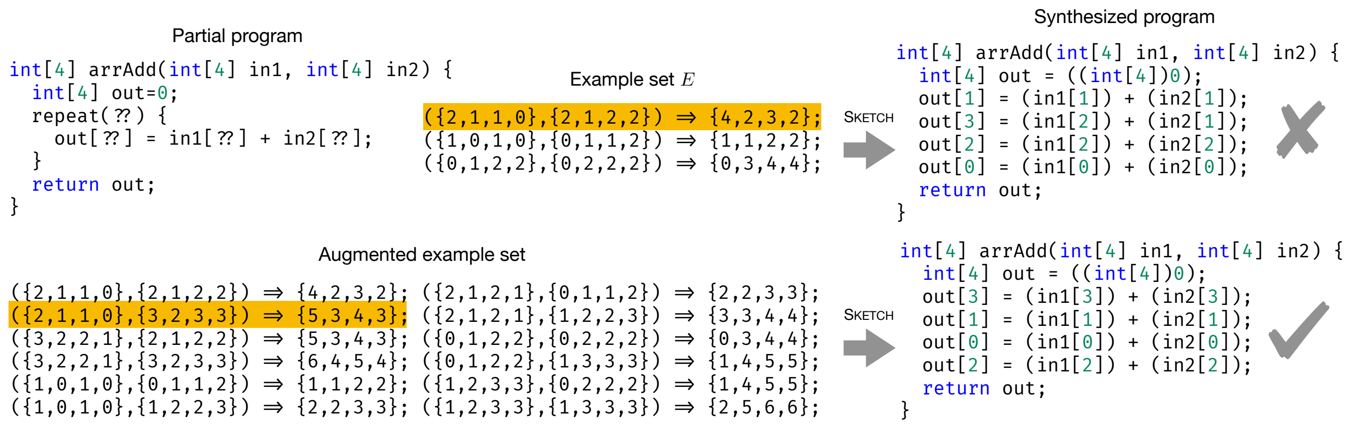
arrayAdd. The top half of Fig. 3 shows a partial program and example set used to synthesize a program that performs the element-wise addition of two arrays in1 and in2. The program generated by Sketch is incorrect. SketchAX applies a value perturbation property to the examples, enabling Sketch to synthesize the correct program. Specifically, if in1 is perturbed by adding d_1 to each of its elements and in2 is perturbed by adding d_2 to each of its elements, each element of the output array should be perturbed by $d_1 + d_2$. The perturbed examples shown in the bottom half of Fig. 3 are obtained using $d_1, d_2 \in \{0, 1\}$.

Remark. Here, we do not discuss the source of relational perturbation properties. Recall that our framework supports three user interfaces to help infer relevant properties. Our procedures for all user interfaces are presented in Sec. 5.

3 Preliminaries

We first define our models of programs and example-based synthesizers.

Programs. The semantics $[\![P]\!]$ of a program P is a function $[\![P]\!] : D_{in} \mapsto D_{out}$ mapping variables over an input domain



```
int[4] arrAdd(int[4] in1, int[4] in2) {
  int[4] out=0;
  repeat(??) {
    out[??] = in1[??] + in2[??];
  }
  return out;
}
```

```
({2,1,1,0},{2,1,2,2}) ⇒ {4,2,3,2};
({1,0,1,0},{0,1,1,2}) ⇒ {1,1,2,2};
({0,1,2,2},{0,2,2,2}) ⇒ {0,3,4,4};
```

```
int[4] arrAdd(int[4] in1, int[4] in2) {
  int[4] out = ((int[4])0);
  out[1] = (in1[1]) + (in2[1]);
  out[3] = (in1[2]) + (in2[1]);
  out[2] = (in1[2]) + (in2[2]);
  out[0] = (in1[0]) + (in2[0]);
  return out;
}
```

```
({2,1,1,0},{2,1,2,2}) ⇒ {4,2,3,2}; ({2,1,2,1},{0,1,1,2}) ⇒ {2,2,3,3};
({2,1,1,0},{3,2,3,3}) ⇒ {5,3,4,3}; ({2,1,2,1},{1,2,2,3}) ⇒ {3,3,4,4};
({3,2,2,1},{2,1,2,2}) ⇒ {5,3,4,3}; ({0,1,2,2},{0,2,2,2}) ⇒ {0,3,4,4};
({3,2,2,1},{3,2,3,3}) ⇒ {6,4,5,4}; ({0,1,2,2},{1,3,3,3}) ⇒ {1,4,5,5};
({1,0,1,0},{0,1,1,2}) ⇒ {1,1,2,2}; ({1,2,3,3},{0,2,2,2}) ⇒ {1,4,5,5};
({1,0,1,0},{1,2,2,3}) ⇒ {2,2,3,3}; ({1,2,3,3},{1,3,3,3}) ⇒ {2,5,6,6};
```

```
int[4] arrAdd(int[4] in1, int[4] in2) {
  int[4] out = ((int[4])0);
  out[3] = (in1[3]) + (in2[3]);
  out[1] = (in1[1]) + (in2[1]);
  out[0] = (in1[0]) + (in2[0]);
  out[2] = (in1[2]) + (in2[2]);
  return out;
}
```

Figure 3. Synthesizing a function to compute the sum of two arrays using Sketch and SketchAX.

D_{in} to variables over an output domain D_{out}. For simplicity of presentation, we assume that D_{in}, D_{out} range over arrays of integers. Our implementation can handle a wider variety of variable domains including scalars, arrays and matrices[1] over Booleans and integers. We use D^n to denote a domain of integer arrays of size n. We say a program P is *consistent* with an input/output (I/O) example (x, y) if $[\![P]\!](x) = y$.

Equivalent programs. Two programs P and P' are *equivalent*, denoted $P \equiv P'$, if P and P' share the same input domain D_{in}, and, $\forall x \in D_{in}.\ [\![P]\!](x) = [\![P']\!](x)$.

Synthesizers. An example-based synthesizer, also sometimes referred to as a *synthesizer*, accepts as input a set E of I/O examples and generates a program P that is consistent with all examples in E. We sometimes refer to I/O examples simply as *examples*. Given a synthesizer S and a set E of examples, we use $S(E)$ to denote the program generated by S[2]. We assume that all user-provided examples are free of error and noise, i.e., all examples are consistent with the implicit specification a user may have in mind.

Some synthesizers are *constraint-based*. Such synthesizers accept constraints in first-order logic (modulo background theories) and use satisfiability modulo theory (SMT) solvers to generate a program that satisfies all constraints. Note that I/O examples can easily be encoded as constraints. Given a set C of constraints, we use $S(C)$ to denote the program generated by a constraint-based synthesizer S in its Max-SMT mode.

In our work, we assume a constraint-based synthesizer can solve a Partial Max-SMT problem. Thus, the synthesizer can accept a set of constraints that are declared to be *hard* (i.e., non-relaxable) or *soft* (i.e., relaxable). Given hard constraints C^{hard} and soft constraints C^{soft}, the synthesizer generates a program, denoted $S(C^{hard}, C^{soft})$, that satisfies all the hard constraints and maximizes the number of satisfied soft constraints.

[1]Matrices are modeled as arrays in our implementation.
[2]We assume that a synthesizer is deterministic. While many synthesizers may execute nondeterministically, they have options to force deterministic behavior. For instance, one can use the '–slv-seed' option for the synthesizer Sketch to force determinism.

4 Relational Perturbation Properties

We now formalize our notion of relational perturbation. We first present a fairly general *parametric* notion of relational perturbation and then present interesting instantiations that are used in our evaluation in Sec. 7.

Perturbation arrays and functions. We consider two classes of perturbation that can be applied to (integer) arrays: structural and value perturbation. A *structural perturbation function* applied to an array changes the positions of the array elements according to a given *structural perturbation array* of indices. A *value perturbation function* applied to an array changes the values of all array elements according to a given *value perturbation array* of parameters. Thus, a structural perturbation function does not modify the values of an array, a value perturbation function does not modify the positions of array elements, and neither perturbation function modifies the size of an array.

Definition 4.1 (Structural perturbation array)**.** A structural perturbation array of size n, Q_n, is an array of indices in D^n: (1) $\forall i \in \{0, \ldots, n-1\}.\ \ Q_n[i] \in \{0, \ldots, n-1\}$ and (2) $\forall i, j \in \{0, \ldots, n-1\}.\ Q_n[i] = Q_n[j] \Rightarrow i = j$.

Definition 4.2 (Structural perturbation function)**.** Let Q_n be a structural perturbation array. A Q_n-structural perturbation function $f^s_{Q_n} : D^n \mapsto D^n$ applied to an array $x \in D^n$ returns an array $x' \in D^n$ such that $\forall i \in \{0, \ldots, n-1\}.\ x[i] = x'[Q_n[i]]$.

Example 4.3. In Fig. 1, the input array $[-1, -3, -2]$ in the highlighted example of E' can be obtained by applying the $[0, 2, 1]$-structural perturbation function to the input array $[-1, -2, -3]$ in the highlighted example of E. We write this as: $f^s_{[0,2,1]}([-1, -2, -3]) = [-1, -3, -2]$.

Example 4.4. The application of $f^s_{[n-1,n-2\ldots,0]}$ to an array $x \in D^n$ returns an array that reverses the elements of x.

The set of all structural perturbation arrays of size n is denoted $\mathcal{Q}_n$. Note that if $n = 1$, then $Q_1 = [0]$ and $f^s_{Q_1}(x) = x$ for any $x \in D^1$. We refer to the *identical* structural perturbation array $[0, 1, \ldots, n-1]$ as id^s_n. Thus, $f^s_{id^s_n}(x) = x$ for any $x \in D^n$. We refer to the structural perturbation array $[k+1, \ldots, n-1, 0, 1, \ldots, k]$, corresponding to a rotation to the right by k positions, as rot^k_n, and the complementary structural perturbation array, corresponding to a rotation to the left by k positions, as rot^{-k}_n.

Definition 4.5 (Value perturbation array)**.** A value perturbation array, $V = [d_1, d_2]$, is an array of rational-valued parameters $d_1, d_2 \in \mathbb{Q}$.

Definition 4.6 (Value perturbation function)**.** Given a value perturbation array $V = [d_1, d_2]$, a V-value perturbation function $f^v_V : D^n \mapsto D^n$ applied to an array $x \in D^n$ returns an array $x' \in D^n$ such that $\forall i \in \{0, \ldots, n-1\}.\ x'[i] = d_1 x[i] + d_2$.

Example 4.7. The application of $f^v_{[2,1]}$ to the array $x = [2, 3, 5, 7]$ yields the array $y = [5, 7, 11, 15]$ and the application of $f^v_{[1/2,-1/2]}$ to y yields x again.

Example 4.8. While the current formalization is limited to single input arrays, we use the example from Figure 3 to illustrate how the formalization extends naturally to multiple input arrays. In Figure 3, the input arrays $([2, 1, 1, 0], [3, 2, 3, 3])$ in the highlighted perturbed example can be obtained by applying a $([1, 0], [1, 1])$-value perturbation function to the input arrays $([2, 1, 1, 0], [2, 1, 2, 2])$ in the highlighted example of E; the first input array is left unchanged and the elements of the second input array are incremented by 1.

Thus, a value perturbation function applies the same affine transformation to all elements of an array. The (infinite) set of all value perturbation arrays is denoted $\mathcal{V}$. We refer to the *identical* value perturbation array $[1, 0]$ as id^v.

Relational perturbation properties. We define *relational perturbation properties* to relate perturbed inputs to corresponding perturbed outputs of programs. We use A and f to denote both structural and value perturbation arrays and functions, respectively. We refer to a perturbation array and perturbation function applied to the input (output) of a program as an *input (output) perturbation array* A_{in} (A_{out}) and an *input (output) perturbation function* $f_{A_{in}}$ ($f_{A_{out}}$), respectively.

Henceforth, we fix the sizes of input and output arrays to be n, m, respectively.

Definition 4.9 (Relational perturbation property)**.** A relational perturbation property R is a tuple $(K_1, K_2, \oplus, \mathcal{A}_{in})$ of a matrix K_1 of rationals, an array K_2 of rationals, an operator $\oplus$ and a set $\mathcal{A}_{in}$ of input perturbation arrays such that: for each $A_{in} \in \mathcal{A}_{in}$, the corresponding output perturbation array $A_{out} = K_1 A_{in} \oplus K_2$[3]. The operator $\oplus \in \{+, +_m\}$ where $+$ is addition and $+_m$ is addition modulo m.

Note that the above definition of a relational perturbation property is very general and can potentially be instantiated in infinitely many ways using its parameters K_1, K_2 and $\mathcal{A}_{in}$. We present two classes of interesting instantiations below that are our evaluation focuses on (Sec. 7). In what follows, $0_{m\times n}$ denotes the zero matrix of size $m \times n$ and $\mathcal{I}_n$ denotes the identity matrix of size n.

Structural relational perturbation properties. These are perturbation properties where both the input and output perturbation is structural. Thus, each A_{in} is a column vector of size n, A_{out} is a column vector of size m, the matrix K_1 is of size $m\times n$, the array K_2 is a column vector of size m, and $\oplus = +_m$.

1. *Permutation invariance.* Permutation invariance specifies that the program output does not change when the elements of the program input (array) are permuted. Formally, for all $Q_n \in \mathcal{Q}_n$, we have $[\![P]\!](x) = [\![P]\!](f^s_{Q_n}(x))$. Permutation invariance is the relational perturbation property $(0_{m\times n}, id^s_m, +_m, \mathcal{Q}_n)$. Note that for all $Q_n \in \mathcal{Q}_n$, $A_{out} = 0_{m\times n}\ Q_n\ +_m\ id^s_m$, i.e., $A_{out} = id^s_m$, as desired.
2. *Permutation preservation.* For this property, we assume that the sizes of input and output arrays are the same, i.e., $n = m$. Permutation preservation specifies that when the elements of the program input are permuted, the elements of the program output are permuted in the same way. Formally, for all $Q_n \in \mathcal{Q}_n$, we have $f^s_{Q_n}([\![P]\!](x)) = [\![P]\!](f^s_{Q_n}(x))$. This can be represented as the relational perturbation property $(\mathcal{I}_n, 0_{n\times 1}, +_n, \mathcal{Q}_n)$.
3. $(k, -k)$*-rotation.* For this property, we also assume that $n = m$. $(k, -k)$-rotation specifies that when the elements of the program input are rotated to the right by k positions, the elements of the program output are rotated to the left by k positions. Formally, for all $k \in \{-(N-1), \ldots, N-1\}$, we have $f^s_{rot^{-k}_n}([\![P]\!](x)) = [\![P]\!](f^s_{rot^k_n}(x))$. This can be represented as the relational perturbation property $(\mathcal{I}_n, k_{n\times 1}, +_n, \{rot^k_n \mid k \in \{-(N-1), \ldots, N-1\}\})$, where $k_{n\times 1}$ is a column vector of size n with all elements equal to k.

Value relational perturbation properties. These are perturbation properties where both the input and output perturbation are value perturbations. Here, A_{in}, A_{out} and the array K_2 are all column vectors of size 2, the matrix K_1 is of size 2×2, and $\oplus = +$.

1. *Value invariance.* Value invariance specifies that for all $V \in \mathcal{V}$, we have $[\![P]\!](x) = [\![P]\!](f^v_V(x))$. Value invariance can be represented by the relational perturbation property $(0_{2\times 2}, id^s_2, +, \mathcal{V})$.

[3]For convenience, we assume arrays are column vectors.

2. *Value preservation.* Value preservation specifies that for all $V \in \mathcal{V}$, we have $f_V^v(\llbracket P \rrbracket(x)) = \llbracket P \rrbracket(f_V^v(x))$ and can be represented as the relational perturbation property $(\mathcal{I}_2, 0_{2\times 1}, +, \mathcal{V})$.

We define two additional value perturbation properties that are used in our evaluation (Sec. 7): $\mathcal{V}_{given}$-value invariance and $\mathcal{V}_{given}$-value preservation. These restrict the focus to a given set $\mathcal{V}_{given}$ of value perturbation arrays, instead of the set $\mathcal{V}$ of all possible value perturbation arrays.

Relational perturbation functions. Relational perturbation functions capture the notion of *applying* a relational perturbation property R to an example set E. Informally, the application of an R-relational perturbation function to E yields a *perturbed* example set E_{pert} obtained by perturbing each example in E according to R.

Definition 4.10 (Relational perturbation function)**.** Given relational perturbation property $R = (K_1, K_2, \oplus, \mathcal{A}_{in})$, an R-relational perturbation function $f_R : (D^n, D^m) \mapsto (D^n, D^m)$ applied to an example set E returns an example set E_{pert} such that $(x', y') \in E_{pert}$ iff there exist $(x, y) \in E$ and $A_{in} \in \mathcal{A}_{in}$ such that $x' = f_{A_{in}}(x)$ and $y' = f_{A_{out}}(y)$ with $A_{out} = K_1 A_{in} \oplus K_2$.

5 Algorithmic Framework

We now present our overall solution framework to improve the generalizability of existing example-based synthesizers. Our framework supports three different user interfaces that differ in the degree of user involvement in identifying suitable relational perturbation properties for an example-based synthesis problem. All our solutions are *synthesizer-agnostic*. The solutions for the first two user interfaces apply to any example-based synthesizer and the solution to the third user interface applies to any constraint-based (example-based) synthesizer. In Sec. 6, we describe how to specialize these solutions to the Sketch synthesizer.

Algorithm 1: Example Augmentation

```
1 procedure PerturbExamples(E, R)
    Input  : $E$: a set of I/O examples
             $R = (k_1, k_2, \oplus, \mathcal{A}_{in})$: a relational
             perturbation property
    Output : $E_{pert}$: a set of I/O examples obtained by
             applying $R$ to $E$
2   $E_{pert} = \emptyset$
3   foreach $(x, y) \in E$ do
4       foreach $A_{in} \in \mathcal{A}_{in}$ do
5           $E_{pert} = E_{pert} \cup \{(f_{A_{in}}(x),\ f_{k_1 A_{in} \oplus k_2}(y))\}$
6   return $E_{pert}$
```

We begin with a procedure that implements the core strategy of our framework: *augment user-provided example sets by applying relational perturbation properties.* Given an example set E and a relational perturbation property R, this simple procedure, shown in Algo. 1, perturbs each example in E by applying R to it according to Def. 4.10.

In what follows, instead of reasoning about parameterized relational perturbation properties (which can possibly be instantiated in infinitely many ways), we restrict our focus to a *finite* set of relational perturbation properties.

5.1 Augmented Synthesis: User Interface I

Algorithm 2: Augmented Synthesis: User Interface I

```
1 procedure AugmentSynthesisI($\underline{E}$, $\underline{\mathcal{R}}$, $S$)
    Input  : $E$, $\mathcal{R}$: as before
             $S$: an example-based synthesizer
    Output : $P$: a program consistent with examples in $E$
2   $E_{aug} = E$
3   for $R \in \mathcal{R}$ do
4       $E_{pert}$ = PerturbExamples($R, E$)
5       $E_{aug} = E_{aug} \cup E_{pert}$
6   return $S(E_{aug})$
```

In User Interface I, the user provides an example set E and a finite set $\mathcal{R}$ of relational perturbation properties. Our solution for this user interface is shown in Algo. 2. We explicitly identify user inputs/interactions in a procedure by underlining them. Note that besides E and $\mathcal{R}$, the procedure also requires as input a synthesizer S. Unlike E and $\mathcal{R}$ which are user-provided inputs (hence, underlined), the synthesizer S is a tunable *parameter* of our framework.

Given these inputs, Algo. 2 generates a program consistent with examples in E. The procedure first uses Algo. 1 to obtain an augmented example set E_{aug} by perturbing the examples in E with all the properties in $\mathcal{R}$. Then, the procedure invokes synthesizer S using E_{aug} to generate the output program.

5.2 Augmented Synthesis: User Interface II

In User Interface II, the user provides an example set E and interacts with our framework to validate/invalidate perturbed examples. The user burden in this case is less than User Interface I — instead of picking applicable relational perturbation properties, the user only needs to examine examples. As before, the user inputs/interactions are underlined in our procedure, Algo. 3, for this user interface. The procedure is additionally parameterized by a synthesizer S, a set of relational perturbation properties $\mathcal{R}$, and the number n of user interactions per property.

For each property in $\mathcal{R}$, Algo. 3 uses Algo. 1 to generate a set E_{pert} of perturbed examples. Then, a set of n randomly chosen perturbed examples from E_{pert} are shown to the user. If the user accepts all n perturbed examples, the example set E is augmented with the perturbed examples E_{pert}. The

Algorithm 3: Augmented Synthesis: User Interface II

```
1  procedure AugmentSynthesisII($\underline{E}, S, \mathcal{R}, n$)
     Input  : $E, S, \mathcal{R}$: as before
              $n$: the number of perturbed examples
              shown to a user
     Output : $P$: a program consistent with examples in $E$
2    $E_{aug} = E$
3    for $R \in \mathcal{R}$ do
4      $E_{pert}$ = PerturbExamples($R, E$)
5      $E_{rand}$ = RandomlyChoose($E_{pert}, n$)
6      if $\underline{\text{UserAccept}(E_{rand})}$ then
7        $E_{aug} = E_{aug} \cup E_{pert}$
8    return $S(E_{aug})$
```

procedure invokes synthesizer S using the final augmented example set E_{aug} to generate the output program.

5.3 Augmented Synthesis: User Interface III

Algorithm 4: Augmented Synthesis: User Interface III

```
1  procedure AugmentSynthesisIII($\underline{E}, S, \mathcal{R}$)
     Input  : $E, \mathcal{R}$: as before
              $S$: a constraint-based synthesizer
     Output : $P$: a program consistent with examples in $E$
2    $C^{hard} = \{P(x) = y \mid (x, y) \in E\}$
3    $C^{soft} = \emptyset$
4    for $R \in \mathcal{R}$ do
5      $E_{pert}$ = PerturbExamples($R, E$)
6      $C = true$;
7      for $(x, y) \in E_{pert}$ do
8        $C = C \wedge (P(x) = y)$
9      $C^{soft} = C^{soft} \cup \{C\}$
10   $\mathcal{L}$ = AllMaxSMTSol($S, C^{hard}, C^{soft}$)
     /* $\mathcal{R}^{\mathcal{L}}$: the set of property sets in $\mathcal{L}$ */
11   return AugmentSynthesisI ($E$, Rank($\mathcal{R}^{\mathcal{L}}$), $S$)
```

In User Interface III, the user only provides an example set E. The user burden in this case is obviously the least among all our user interfaces. In fact, there is no additional burden on the user beyond a standard PBE setting. Not surprisingly, this user interface is the most challenging for our framework as we need to automatically *infer* relevant relational perturbation properties without any help from a user. Our solution, based on a Partial MAX-SMT formulation, is shown in Algo. 3. Besides the (underlined) user-provided example set, the procedure is parameterized by a synthesizer S and a set of relational perturbation properties $\mathcal{R}$. We require S to be a constraint-based synthesizer. Thus, S can solve a Partial MAX-SMT problem.

For each example in E, the procedure generates a corresponding hard constraint. For each property $R \in \mathcal{R}$, the procedure generates a soft constraint corresponding to the set of perturbed examples obtaining by applying R to E (using Algo. 1). Once all constraints are generated, we have a Partial MAX-SMT synthesis problem defined by the tuple (S, C^{hard}, C^{soft}) A solution $(P, \mathcal{R}^*)$ to this partial MAX-SMT synthesis problem consists of a program P, synthesized by S, which is consistent with the set of all examples in E (i.e., C^{hard}) and all examples perturbed according to some maximal subset of properties $\bar{\mathcal{R}} \subseteq 2^{\mathcal{R}}$ (corresponding to a maximally satisfiable set of soft constraints in C^{soft}). Note that, in general, there can be multiple such solutions, say $\{(P_1, \bar{\mathcal{R}}_1), \ldots, (P_t, \bar{\mathcal{R}}_t)\}$; let us denote this set by $\mathcal{L}$. If Algo. 4 were to simply return $S(C^{hard}, C^{soft})$, this would be a program P corresponding to an arbitrary solution $(P, \bar{\mathcal{R}})$ from $\mathcal{L}$ (based on the search strategy of S). In particular, P may not be the most generalizable program and $\bar{\mathcal{R}}$ may not be the most suitable property set for the given example-based synthesis problem. While it is not clear how to formally define optimality of solutions to the Partial MAX-SMT synthesis problem, Algo. 4 uses a more sophisticated approach than simply returning $S(C^{hard}, C^{soft})$.

First, Algo. 4 uses a procedure AllMaxSMTSol to obtain the entire set $\mathcal{L}$ of Partial MAX-SMT synthesis solutions (Line 10). Let $\mathcal{R}^{\mathcal{L}}$ denote the set of property sets in $\mathcal{L}$. In Line 11, Algo. 4 uses a procedure Rank to obtain the property set in $\mathcal{R}^{\mathcal{L}}$ ranked highest by a ranking function and invokes Algo. 2 with this highest ranked property set.

The procedures AllMaxSMTSol and Rank can be instantiated in many ways. We describe our specific implementations in the next section.

5.4 Correctness

An example-based synthesizer S is sound if: whenever S generates a program P from a set E of examples, P is guaranteed to be consistent with all examples in E. An example-based synthesizer S is complete if: whenever there exists a program in S's hypothesis space consistent with examples in a given example set E, S can always generate such a program.

Recall that we assume that a user does not make mistakes: all user-provided/validated examples are free of error.

Theorem 5.1. *The synthesis procedures in Algo. 2, Algo. 3 and Algo. 4 are sound and complete if the synthesizer S is sound and complete.*

Note that we are unable to provide any formal guarantees about the generalizability of the programs synthesized by our procedures. However, as we will see in Sec. 7, all our procedures can significantly improve the generalizabity of the SKETCH synthesizer.

```
int[N] F(int[N] in){
  ?? // holes in unknown program
  return out;
}
harness void userProvidedE(E){
  for (in, out) ∈ E:
      assert out == F(in);
}
harness void augmentedE(E_pert){
  for (in, out) ∈ E_pert:
      assert out == F(in);
}
```

Figure 4. Sketch encoding for Algo. 1.

6 SketchAX

In this section, we describe the key components of the specialization, SketchAX, of our framework to the Sketch synthesizer. Besides the basic Sketch encoding of our algorithms, we present an efficient alternative to `AugmentSynthesisIII` and a novel procedure for learning a ranking function over property sets from *training data*.

Basic Sketch encoding. The main idea of the Sketch encoding for all our algorithms (see Fig. 4) is to use the `harness` function in Sketch to impose I/O constraints, corresponding to user-provided and augmented examples, as `assert` statements.

Implementation of `AugmentSynthesisIII`. When using an exact encoding of Algo. 4 in Sketch, we found that Sketch often struggles to complete the difficult Partial Max-SMT optimization problem within a specified time bound (even for returning one solution). Hence, we encode a simple greedy procedure for solving the maximization constraint: instead of considering all properties in $\mathcal{R}$ at once, the procedure performs a greedy search over subsets of properties in $\mathcal{R}$ of increasing sizes. The procedure marks a property set as *satisfiable* if `AugmentSynthesisI` can successfully synthesize a program using the property set and Sketch. The procedure maintains largest satisfiable subsets of properties until no subset can be expanded any further, or the procedure exceeds a time bound. Note that the procedure generates all solutions to the Partial Max-SMT that can be computed within the time bound.

Learning a ranking function. Algo. 5 describes our procedure for learning a ranking function over property sets in $\mathcal{R}$ from a training set. The training set consists of a list of *successful augmented synthesis instances from User Interface I*; each instance contains an I/O example set, a *correct* set of relational perturbation properties, and a *correct* program synthesized by `AugmentSynthesisI` using the properties; correctness of the program and property set is verified using a complete functional specification. The procedure accepts the training set as three separate lists, $\mathbb{E}^T$, $\mathbb{P}^T$ and $\mathbb{R}^T$ as shown and outputs a ranked list $\mathbb{A}$ of property sets. Note

Algorithm 5: Learning the Ranking Function

```
1  procedure LearnRank($\mathbb{E}^T, \mathbb{P}^T, \mathbb{R}^T, S$)
       Input  : $\mathbb{E}^T$: a list of $n$ I/O example sets
                $\mathbb{P}^T$: a list of $n$ correct synthesized programs
                $\mathbb{R}^T$: a list of $n$ correct property sets
                $S$: an example-based synthesizer
       Output : $\mathbb{A}$: a ranked list of property sets
       /* $\mathcal{D}$ is a dictionary */
2      $\mathcal{D} = \{\}$
3      for $i \in [1, 2, \ldots, n]$ do
4          $\mathcal{L}$ = AllMaxSMTSynthesis($\mathbb{E}^T[i]$, $S$, $\mathcal{R}$)
5          $\mathcal{L}^+ = \{(P, \bar{\mathcal{R}}) \mid (P, \bar{\mathcal{R}}) \in \mathcal{L} \wedge \bar{\mathcal{R}} \subseteq \mathbb{R}^T[i]\}$
6          $\mathcal{L}^- = \mathcal{L} \setminus \mathcal{L}^+$
7          for $(P_j, \bar{\mathcal{R}}_j) \in \mathcal{L}^+$ do
8              for $(P_k, \bar{\mathcal{R}}_k) \in \mathcal{L}^-$ do
9                  if "$\bar{\mathcal{R}}_j > \bar{\mathcal{R}}_k$" $\notin L$ then
10                     $\mathcal{D}$["$\bar{\mathcal{R}}_j > \bar{\mathcal{R}}_k$"] = 0
11                 if $P_j \equiv \mathbb{P}^T[i]$ then
12                     $\mathcal{D}$["$\bar{\mathcal{R}}_j > \bar{\mathcal{R}}_k$"]+ = 1
13     foreach '$\bar{\mathcal{R}}_j > \bar{\mathcal{R}}_k$" $\in \mathcal{D}$ do
14         if "$\bar{\mathcal{R}}_k > \bar{\mathcal{R}}_j$" $\in \mathcal{D}$ then
15             if $\mathcal{D}$["$\bar{\mathcal{R}}_j > \bar{\mathcal{R}}_k$"] $\geq \mathcal{D}$["$\bar{\mathcal{R}}_k > \bar{\mathcal{R}}_j$"] then
16                 delete "$\bar{\mathcal{R}}_k > \bar{\mathcal{R}}_j$" from $\mathcal{D}$
17             else
18                 delete "$\bar{\mathcal{R}}_j > \bar{\mathcal{R}}_k$" from $\mathcal{D}$
19     $\mathbb{A}$ = RankProperties($\mathcal{D}, \mathcal{R}$)
20     return $\mathbb{A}$
```

that `LearnRank` can be parameterized by any example-based synthesizer S (it uses Sketch in SketchAX).

The ranked list $\mathbb{A}$ is computed from a dictionary $\mathcal{D}$, where a key is a string (of the form "$\mathcal{R}_1 > \mathcal{R}_2''$) representing an ordered pair of property sets and the value is the number of *successful instances* of that ordering in the training set. Let `AllMaxSMTSynthesis` denote a version of the procedure `AugmentSynthesisIII` that is identical till Line 10 and returns the set $\mathcal{L}$ of all solutions to a Partial Max-SMT synthesis problem (computed in Line 10). For each example set in the training set, `LearnRank` updates the dictionary in Lines 4–12 as follows. Given the i^{th} example set $\mathbb{E}^T[i]$, `AllMaxSMTSynthesis` is used to infer the set $\mathcal{L}$ of all solutions $(P, \bar{\mathcal{R}})$ using the example set. If the property set $\bar{\mathcal{R}}$ of a solution is a subset of the corresponding correct set of properties in $\mathbb{R}^T[i]$, the solution is added to the set $\mathcal{L}^+$ of *accepted solutions*. If not, the solution is added to the set $\mathcal{L}^-$ of *rejected solutions*. The dictionary is updated by creating entries with keys that order every property set in $\mathcal{L}^+$ above every property set in $\mathcal{L}^-$ (Lines 7–10). Moreover, each time the program in an accepted solution is found to be semantically equivalent to the known correct program $\mathcal{P}^T[i]$, the value for the corresponding entry in the dictionary is incremented (Lines 11–12).

Finally, a simple processing in Lines 14–18 ensures consistency of the dictionary by making sure the ordering relation is anti-symmetric (the ordering with the lower value is deleted). Note that the dictionary encodes a partial order between property sets as not all property sets may be ordered. The procedure `RankProperties` returns a ranked list of property sets over $\mathcal{R}$ compatible with the dictionary.

7 Evaluation

Our comprehensive evaluation of SketchAX investigates the improvement over Sketch in synthesis of correct benchmarks and the run-time performance of our algorithms for all three user interfaces. We also take a closer look at the sensitivity to parameters such as training set size and example set size. In what follows, the implementations in SketchAX of the algorithms for the three user interfaces are denoted SketchAX I, SketchAX II and SketchAX III, respectively.

Experimental setup We focus on the following set of relational perturbation properties as they suffice for the benchmarks we considered — p_1: permutation invariance, p_2: permutation preservation, p_3: $(k, -k)$-rotation, p_4: $\mathcal{V}_{add}$-value preservation, p_5: $\mathcal{V}_{mult}$-value preservation and p_6: permutation transposition. Here, $\mathcal{V}_{add}$ is $\{[0, d] \mid d \in \{1, \ldots, 10\}\}$ and $\mathcal{V}_{mult}$ is $\{[d, 0] \mid d \in \{2, \ldots, 10\}\}$. Thus, property p_4 perturbs inputs/outputs by adding $d \in \{1, \ldots, 10\}$ to all elements and property p_5 perturbs inputs/outputs by multiplying all elements with $d \in \{2, \ldots, 10\}$. Property p_6, illustrated in Fig. 2, applies an identical permutation to the rows and columns of input and output matrices, respectively.

We analyze the effectiveness of SketchAX on 168 benchmarks from *Sketch Repositories* [1]. Each benchmark was automatically selected using a script based on the following requirements: (1) there is a complete functional specification for the benchmark; (2) the benchmark contains at least one hole; (3) Sketch can synthesize a program from the benchmark; (4) and the input/output of the benchmark are of types bit/int or their arrays. Based on the input/output types, we further classify the 168 benchmarks into 3 classes: *Bit* benchmarks which only use bits/bit-vectors; *Int* benchmarks which only use int/int arrays; and *Mixed* benchmarks which use both bits/bit-vectors and int/int arrays.

We leverage the complete functional specifications of the benchmarks to check the correctness of synthesized programs (using the keyword `implements` in Sketch for checking program equivalence).

In our default setting, we evaluate the success rate of Sketch and our algorithms over 10 runs for each benchmark, yielding *1680 synthesis instances* in total. Each run uses a different set of 3 I/O examples, randomly generated from the complete functional specification. We set a timeout of 5 minutes for synthesis-solving across all experiments. For the perturbation of an I/O example with a relational perturbation property (Line 5 in `AugmentSynthesisI`), we set a timeout of 5 seconds and an upper-bound of 128 perturbed examples to ensure Sketch terminates within a reasonable time. On average, we generated 160 perturbed examples for each benchmark.

We use Sketch version 1.7.5. We ran our experiments on shared servers equipped with Intel E5320@1.86GHz CPU and 8GB RAM.

7.1 SketchAX I

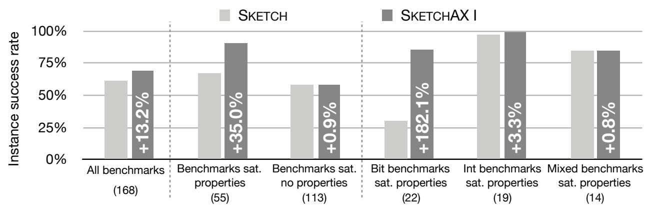


Figure 5. Instance success rate of Sketch and SketchAX I.

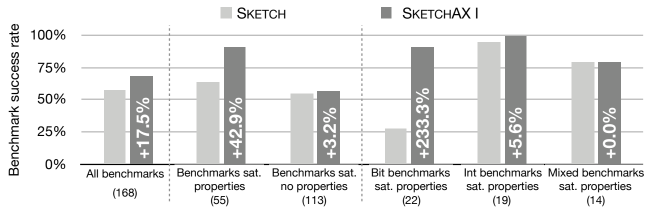


Figure 6. Benchmark success rate of Sketch and SketchAX I.

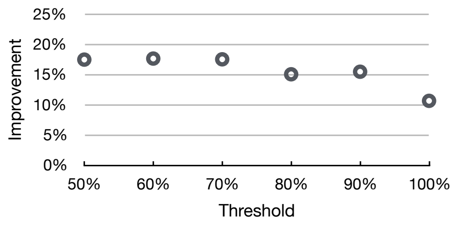


Figure 7. Improvement of benchmark success rate w.r.t. varying thresholds for defining successful benchmark synthesis.

Figures 5 and 6 summarize the results of our synthesis experiments with Sketch and SketchAX I. For each benchmark, the applicable relational perturbation properties (from $p_1, \ldots, p_6$) are manually chosen by us. Fig. 5 shows the *instance success rate* for different benchmark categories. Recall that there are 10 synthesis instances per benchmark. The instance success rate is the percentage of synthesis instances that yield correct synthesized programs. The numbers below each benchmark category on the x-axis indicate the number of benchmarks in that category. Fig. 6 shows the *benchmark success rate* for different benchmark categories. A benchmark

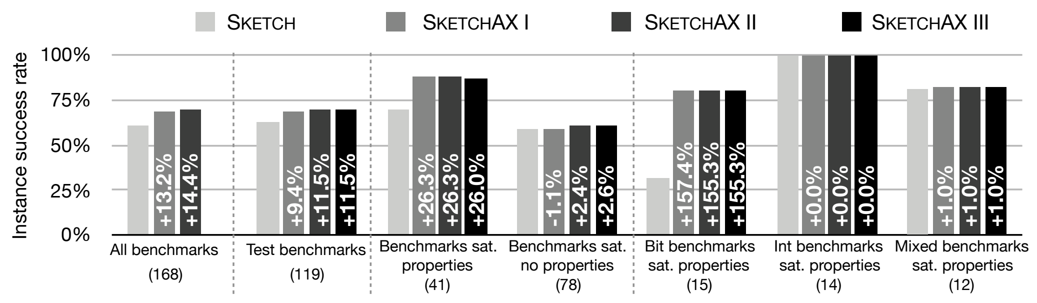


Figure 8. Instance success rate of Sketch and SketchAX algorithms.

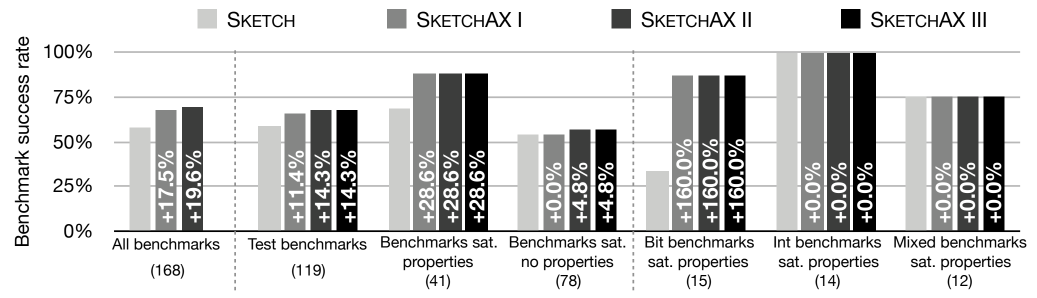


Figure 9. Benchmark success rate of Sketch and SketchAX algorithms.

synthesis is declared successful if at least 70% if its synthesis instances yield correct synthesized programs. The benchmark success rate is then simply computed as the percentage of benchmarks whose synthesis is successful. We later investigate other thresholds for defining a successful benchmark synthesis (Fig. 7).

Let us first take a closer look at Fig. 5. The numbers within the SketchAX I bars indicate the improvement over the instance success rate of Sketch with SketchAX I. The overall improvement in the instance success rate of Sketch with SketchAX I is **13.2**%. The improvement is significantly higher (**35**%) for benchmarks which satisfy at least one of the relational perturbation properties $p_1, \ldots, p_6$, thereby validating our fundamental hypothesis. The improvement is astronomical (**182.1**%) for Bit benchmarks satisfying some perturbation properties, a category in which Sketch does not perform too well by itself. The improvement for Int benchmarks satisfying some properties is small (**3.3**%) as Sketch performs really well in this category by itself. The improvement for benchmarks that do not satisfy any properties and for Mixed benchmarks is negligible. It is important to note, though, that even for benchmarks that do not satisfy any properties, SketchAX's success rate does not fall below that of Sketch.

The improvements in benchmark success rates, shown in Fig. 6, are higher: **17.5**% for all benchmarks, **233**% for Bit benchmarks satisfying some properties, and **5.6**% for Int benchmarks satisfying some properties.

In Fig. 7, we investigate the effect of different thresholds for defining successful benchmark synthesis. The y-axis tracks the improvement in benchmark success rate of SketchAX I over Sketch for all benchmarks. Notice that for all thresholds, SketchAX performs better than Sketch, typically by more than 15%. However, for the 100% threshold, the improvement drops to 10.7%. Since our example sets are randomly-generated, there are often 1-2 example sets of particularly poor quality, for which SketchAX I fails to synthesize correct programs.

Henceforth, we use 70% as the default threshold to measure benchmark success rate.

7.2 SketchAX II

Figures 8 and 9 summarize the results of our synthesis experiments with Sketch and all SketchAX algorithms. Note that there is an extra category — Test benchmarks — in these figures and the numbers associated with the other categories have changed from Figures 5 and 6. As mentioned in Sec. 6, we need to partition the benchmarks to train and test the ranking function. Hence, we randomly choose 49 of the 168 benchmarks to learn our ranking function, and test SketchAX III on the rest. To keep the comparison fair, we use the same test benchmarks to also evaluate the other algorithms. We investigate other choices for partitioning the benchmarks later (see Fig. 10).

Figures 8 and 9 show that SketchAX II, which employs a user to validate 1 perturbed example per property, has a similar performance as SketchAX I across all categories.

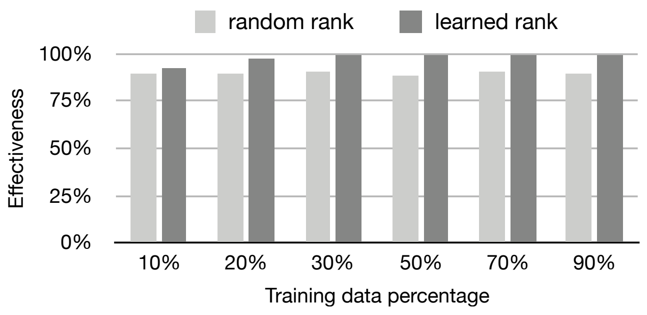


Figure 10. Effectiveness of ranking function for varying training set sizes

Further, observe that for the benchmarks which do not satisfy any properties, SketchAX II performs slightly better than SketchAX I, leading to a small overall improvement in its success rate across all benchmarks. There is an interesting explanation for this. For some synthesis instances, some relational perturbation properties hold on the given example set even though the properties don't hold for the program in general. Thus, the user validates the resulting perturbed examples and SketchAX II applies the properties to successfully augment the example set and the synthesis. In contrast, SketchAX I does not apply such properties to the example set and hence, fails to augment the synthesis in these cases.

We also tested SketchAX II by having the user validate 2 and 3 perturbed examples per property and found that the results were similar.

7.3 SketchAX III

The noteworthy improvements in success rates over Sketch we have discussed so far have been for SketchAX with User Interfaces I and II, where the user plays an active role in providing or helping infer an applicable set of relational perturbation properties. The real testament to SketchAX's ability to augment Sketch lies in the success rates of SketchAX III in Figures 8 and 9. SketchAX III performs similar to Sketch in most categories, with a small overall improvement for all (test) benchmarks. The reason for this small overall improvement is the same as for SketchAX II.

Effectiveness of ranking function and impact of training set size. The learned ranking function is clearly a key contributor to the success of SketchAX III. We take a closer look in Fig. 10 at the *effectiveness* of a learned ranking function over the Bit benchmarks. We focus on this benchmark category as changes in effectiveness are more obvious (than in categories where the improvement of SketchAX III over Sketch is less). We define the effectiveness of a ranking function as its success rate divided by the (maximal) success rate of `AllMaxSMTSynthesis` (see Algo. 5). Recall that `AllMaxSMTSynthesis` returns the set of all solutions to a

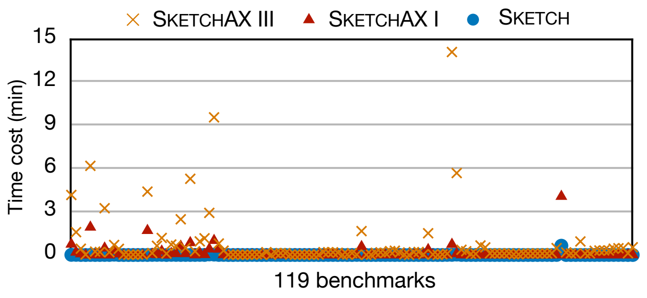


Figure 11. Time cost of Sketch, SketchAX I and SketchAX III.

Table 1. Statistical view of time cost (in seconds).

	Sketch	SketchAX I	SketchAX III
Average	1.52	9.14	64.79
1st quartile	1.02	1.12	3.19
Median	1.08	1.34	8.88
3rd quartile	1.25	2.13	28.06

Partial Max-SMT synthesis problem. An instance of using `AllMaxSMTSynthesis` is declared successful if *any* of the solutions it generates contains a correct program. Obviously, this give us an upper-bound for the success rate of SketchAX III, regardless of what ranking function we use. Fig. 10 compares the effectiveness of our learned ranking strategy with that of a ranking function that randomly choses a property set from the set of all Partial Max-SMT solutions. Our learned ranking function is clearly more effective.

We also evaluate the impact of the size of the training set on the effectiveness of the ranking strategy. The x-axis in Fig. 10 indicates the training set size as a percentage of the 168 benchmarks, with the training benchmarks chosen randomly. Fig. 10 demonstrates that a training set that uses 30% of the benchmarks yields a ranking function with effectiveness 99.2%. This justifies our default choice of the training set size.

Inference of correct property sets. Since SketchAX III infers relevant property sets to use to augment examples, we examine its *inference accuracy*. For the 41 test benchmarks that satisfy some properties, this inference accuracy tracks the percentage of synthesis instances for which SketchAX III inferred property sets that are subsets of the correct property sets. The inference accuracy is **93.3**%, **100**% and **94.2**% for Bit, Int and Mixed benchmarks, respectively, and **95.9**% overall. Thus, the inference accuracy of SketchAX III is quite high.

7.4 Time cost of SketchAX

Fig. 11 shows the total synthesis time taken by Sketch and SketchAX I and SketchAX III on each of the 119 test benchmarks. We exclude SketchAX II as, barring the user interactions (whose time cost is hard to estimate), the computations

Table 2. Instance success rate (%) with varying example set size

	(78) Bit benchmarks					(15) Bit benchmarks sat. properties					(63) Bit benchmarks sat. no properties				
	1 ex.	2 ex.	3 ex.	4 ex.	5 ex.	1 ex.	2 ex.	3 ex.	4 ex.	5 ex.	1 ex.	2 ex.	3 ex.	4 ex.	5 ex.
Sketch	32.2	45.8	56.8	61.8	65.4	18.0	22.7	31.3	35.3	40.7	35.6	51.3	62.9	68.1	71.3
SketchAX I	39.1	55.1	65.6	71.2	73.9	45.3	68.7	80.7	84.0	84.0	37.6	51.9	62.1	68.1	71.4
Improvement	21.5	20.5	15.6	15.2	12.9	151.9	202.9	157.5	137.7	106.6	5.8	1.2	-1.3	0.0	0.2
SketchAX II	40.5	56.4	67.7	72.1	74.2	44.7	68.7	80.7	84.0	84.0	39.5	53.5	64.6	69.2	71.9
Improvement	25.9	23.3	19.2	16.6	13.5	148.2	202.9	157.5	137.7	106.6	11.2	4.3	2.8	1.6	0.9
SketchAX III	42.4	59.1	67.7	71.8	74.1	45.3	68.7	80.0	82.0	83.3	41.7	56.8	64.8	69.4	71.9
Improvement	31.9	29.1	19.2	16.2	13.3	151.9	202.9	155.3	132.1	104.9	17.4	10.8	3.0	1.9	0.9

in SketchAX II are almost identical to those in SketchAX I. We also exclude the disproportionately high time (42 minutes) taken by SketchAX III on on one benchmark as it made it harder to see the time patterns in the rest of the benchmarks. Observe that for most benchmarks, SketchAX I and SketchAX III took time similar to that taken by Sketch. The statistical results in Table 1 provide a more fine-grained view of the time performance. While the average times taken by SketchAX I and SketchAX III are noticeably higher than that taken by Sketch, their times for the 3 quartiles are significantly lower and quite reasonable. Notice that SketchAX I closely matches the times taken by Sketch on the 1st and 2nd quartile. And for 75% of the benchmarks (3rd quartile), the time taken by SketchAX I is 2.13 seconds. As for SketchAX III, 25% of the benchmarks (1st quartile) were synthesized within 3.19s and half of the benchmarks (median) were tackled within 8.88 seconds.

7.5 Sensitivity of SketchAX to example set size

To illustrate how the size of the user-provided example set affects the success rate of SketchAX algorithms, we ran the algorithms with 1 to 5 examples for the test Bit benchmarks (see Table 2). As expected, more I/O examples can improve the success rates of all tools/algorithms: (from about **30%** to about **65%** for Sketch and from about **40%** to about **74%** for SketchAX algorithms). In general, the improvement due to augmented synthesis increases as the number of examples decreases. However, for benchmarks satisfying some relational perturbation properties, the improvement starts going down when the number of examples decreases below a certain threshold (2 in our case). This is because the number of initial examples is too small to reliably augment.

8 Related Work

Data augmentation in machine learning. Deep learning techniques use data augmentation as a common technique for improving machine learning classifiers [12, 18]. For instance, for learning a classifier on a set of input images, they generate new images by applying label-preserving transformations (e.g. image translation, horizontal reflections, or altering RGB channel intensities). This not only provides more training data for large deep learning models, but also enables the model to learn certain input invariance properties, thereby improving generalization on unseen data. In contrast, our algorithms for PBE synthesis learn the desired program from a small set of examples. Instead of limiting ourselves only to label-preserving transformations, our framework also supports perturbations where the labels (outputs) can change with respect to the input changes, e.g. the permutation preservation and value preservation perturbations.
Handling ambiguity in PBE. Besides using highly structured DSLs [2, 24] to place a syntactic bias on the hypothesis space, many PBE synthesizers use a ranking function that aims to score consistent programs by their ability to generalize. The ranking function is either manually designed [8] or learnt from data [21]. Another approach gathers additional information from the user to disambiguate the program space [15], e.g., by creating distinguishing inputs [11] or abstract examples [5]. Raychev et al. [17] present a feedback loop that identifies and discards potentially incorrect examples. Unlike these approaches, our approach handles ambiguity by placing a semantic bias on the hypothesis space using relational perturbation properties to automatically augment the example sets.
Programming By Examples. PBE techniques have been successfully developed for various domains: string transformations [8, 19], data structure manipulations [6, 22], number transformations [20], parser synthesis [13], map-reduce style distributed programs [23], web data integrations [10]. Recent papers [4, 16] use deep learning to automatically generate PBE systems. Most of these PBE systems generate programs that are consistent with a user-provided set of examples. Systems such as BlinkFill [19] also take into account additional specifications (besides examples) from spreadsheets. Our approach can complement many existing synthesizers given corresponding perturbation properties in different domains.
Relational program synthesis. Recent work on relational program synthesis [25] seeks to synthesize programs from complete relational specifications. In contrast, we use a class of relational properties to augment program synthesis from incomplete example-based specification.

9 Conclusion

We proposed a new approach to address the ambiguity/-generalibility issue in PBE based on the idea of example augmentation using relational perturbation properties. We presented synthesizer-agnostic solutions for three user interfaces and demonstrated the effectiveness of our approach in significantly boosting the performance of the Sketch synthesizer. As a next step, we plan to investigate richer classes of relational properties in diverse domains and apply our approach to multiple PBE synthesizers.

References

[1] [n. d.]. Sketch Samples Repository. https://bitbucket.org/gatoatigrado/sketch-frontend/src.

[2] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *FMCAD*. IEEE, 1–8.

[3] Alessandro Cimatti, Anders Franzén, Alberto Griggio, Roberto Sebastiani, and Cristian Stenico. 2010. Satisfiability Modulo the Theory of Costs: Foundations and Applications. In *Tools and Algorithms for the Construction and Analysis of Systems*. 99–113.

[4] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. RobustFill: Neural Program Learning under Noisy I/O. In *ICML*. 990–998.

[5] Dana Drachsler-Cohen, Sharon Shoham, and Eran Yahav. 2017. Synthesis with Abstract Examples. In *CAV*. 254–278.

[6] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *PLDI*. 229–239.

[7] Sumit Gulwani. 2016. Programming by Examples: Applications, Algorithms, and Ambiguity Resolution. In *Proceedings of the 8th International Joint Conference on Automated Reasoning - Volume 9706*. Springer-Verlag, Berlin, Heidelberg, 9–14. https://doi.org/10.1007/978-3-319-40229-1_2

[8] Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet data manipulation using examples. *Commun. ACM* 55, 8 (2012), 97–105.

[9] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends in Programming Languages* 4, 1-2 (2017), 1–119.

[10] Jeevana Priya Inala and Rishabh Singh. 2018. WebRelate: integrating web data with spreadsheets using examples. *PACMPL* 2, POPL (2018), 2:1–2:28.

[11] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *ICSE*. 215–224.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*. 1106–1114.

[13] Alan Leung, John Sarracino, and Sorin Lerner. 2015. Interactive parser synthesis by example. In *PLDI*. 565–574.

[14] Henry Lieberman. 2000. Programming by Example: Introduction. *Commun. ACM* 43, 3 (2000), 72–74.

[15] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin G. Zorn, and Sumit Gulwani. 2015. User Interaction Models for Disambiguation in Programming by Example. In *UIST*. 291–301.

[16] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2017. Neuro-Symbolic Program Synthesis. In *ICLR*.

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

[18] Patrice Y. Simard, David Steinkraus, and John C. Platt. 2003. Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis. In *ICDAR*. 958–962.

[19] Rishabh Singh. 2016. BlinkFill: Semi-supervised Programming By Example for Syntactic String Transformations. *PVLDB* 9, 10 (2016), 816–827.

[20] Rishabh Singh and Sumit Gulwani. 2012. Synthesizing Number Transformations from Input-Output Examples. In *CAV*. 634–651.

[21] Rishabh Singh and Sumit Gulwani. 2015. Predicting a Correct Program in Programming by Example. In *CAV*. 398–414.

[22] Rishabh Singh and Armando Solar-Lezama. 2011. Synthesizing data structure manipulations from storyboards. In *FSE*. 289–299.

[23] Calvin Smith and Aws Albarghouthi. 2016. MapReduce program synthesis. In *PLDI*. 326–340.

[24] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, USA, 404–415. https://doi.org/10.1145/1168857.1168907

[25] Yuepeng Wang, Xinyu Wang, and Isil Dillig. 2018. Relational Program Synthesis. *Proc. ACM Program. Lang.* 2, OOPSLA (Oct. 2018), 155:1–155:27. https://doi.org/10.1145/3276525