Modeling Black-Box Components using Probabalistic Synthesis

Anonymous Author(s)

Abstract

This paper is concerned with synthesizing programs based on *black-box* oracles: we are interested in the case where there exists an executable implementation of a component or library, but its internal structure is unknown. We are provided with just an API or function signature, and aim to synthesize a program with equivalent behavior.

To attack this problem, we detail Presyn: a program synthesizer designed for flexible interoperation with existing programs and compiler toolchains. Presyn uses high-level imperative control-flow structures and a pair of cooperating predictive models to efficiently narrow the space of potential programs. These models can be trained effectively on small corpora of synthesized examples.

We evaluate Presyn against five leading program synthesizers on a collection of 112 synthesis benchmarks collated from previous studies and real-world software libraries. We show that Presyn is able to synthesize a wider range of programs than each of them with less human input. We demonstrate the application of our approach to real-world code and software engineering problems with two case studies: accelerator library porting and detection of duplicated library reimplementations.

Keywords program synthesis,black box oracle,probabilistic model

1 Introduction

Modeling and understanding the behavior of software components is a key issue in software engineering [15]. It has been used to discover similarity between libraries [56], rejuvenate code [36] and match accelerators to software [20]. While a formal model of a component may be available, it is often supplied as low-level binary or not at all [36]. Our goal is to model such "black-box" software components, in a form suitable for interoperation with real world code using *program synthesis*.

For the sake of wide applicability, we make minimal assumptions about the component being modeled: we require only that it exists in an executable form and has a known type signature. This paper therefore addresses the problem of program synthesis based on the observed behavior of an oracle performing an existing, but unknown computation [32]. We have no knowledge of the oracle's internal structure, but capturing its behavior in the form of input-output (IO) examples is inexpensive.

This problem falls under the domain of *programming by example* which has received considerable interest from industry [28]. Here, the aim is to synthesize a program from user provided examples [10, 27]. Our approach differs from the standard formulation in that IO examples are effectively free, as we do not rely on the user.

Our approach is distinct to much prior work, where their aim is to generate provably correct programs with respect to a formal specification (typically making use of counter-examples and SMT solvers) [7, 14, 18]. Instead, it is closer in spirit to neural synthesis approaches [11, 37, 42], where IO examples *are* the specification of a task. These schemes, however, are currently limited in the problem domains that can be addressed. [45].

Like previous work which attempts to synthesize without a formal specification, we cannot guarantee correctness of the generated program [10]. However, testing over a large set of examples can provide observational equality, which is widely accepted as a sufficiently strong guarantee [19] in the absence of formal proof.

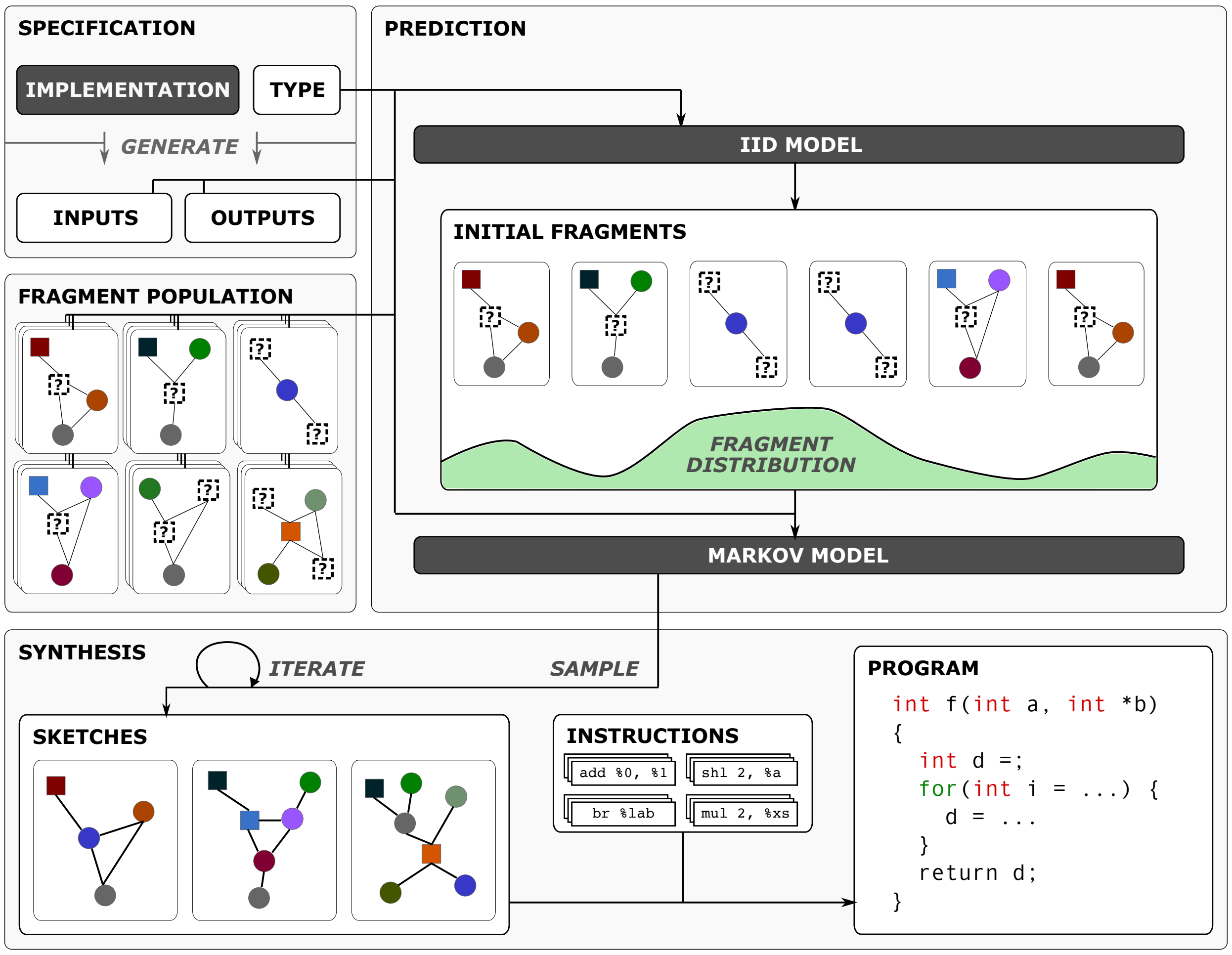
Sketching Our approach uses ideas from *sketching* program synthesis [54]: a high level partial structure, or sketch, of the target program allows for efficient search through a space of potential solutions. While many techniques use externally provided sketches [53], our scheme uses a two-phase synthesis process [57] where sketches are constructed by the synthesizer based on the problem specification (in our case, a type signature and IO examples).

However, IO examples alone are not sufficient to synthesize programs that use complex components. We use prior observations of synthesized program structure to build a probability distribution over fragments of sketches given a type signature and a set of IO examples. Specifically, we develop two models: IID and Markov.

The first, IID, assumes independence among sketch fragments (**I**ndependent and **I**dentically **D**istributed), while the second assumes the next fragment is dependent on the current one (i.e. a Markov model). Using this approach, we synthesize a wide range of complex components from black-box specifications.

There is a large body of related work in this diverse area. and providing a fair comparison is a challenge [30, 41]. In this paper we attempt a fair, systematic and reproducible

Conference'17, July 2017, Washington, DC, USA
2020.



```
int f(int a, int *b)
{
  int d =;
  for(int i = ...) {
    d = ...
  }
  return d;
}
```

Figure 1. A summary of Presyn's implementation. Fragment distributions are learned by IID and Markov models using the problem specification and initial fragment population. Sketches are then sampled and synthesized into executable programs.

evaluation of representative state-of the art existing schemes. For that reason, our results and software will be publicly available at [6]

As our approach is driven by a probabilistic model of sketches, we analyze the learned distributions and the resulting insights into program structure. We then evaluate the use of black-box synthesis in two application areas: rejuvenating legacy scientific code and detecting reimplementations of library code.

1.1 Contributions

We implement a novel program synthesizer (Presyn) that uses lightweight probabilistic models to efficiently search for solutions in varied problem domains. We evaluate Presyn against five other program synthesizers from different traditions: neural SketchAdapt [37], functional λ^2 [25], imperative Simpl [53], type-directed TypeDirect [20] and genetic Makespeare [49],

This paper makes the following main contributions:

- Probabilistic synthesis based on corpus priors with broader synthesis results than existing work
- An extensive and systematic evaluation of example-based synthesis from different research domains
- An analysis of sketch distributions across synthesized programs
- Two case studies showing uses of black box synthesis for software engineering

2 Overview

Figure 1 gives an overview of our program synthesizer, Presyn. It consists of four primary components: specification and example construction; fragment population; probabilistic fragment models and program synthesis.

Given a function signature, we can generate inputs by sampling random values from the domain of each input parameter. As we have an executable implementation of the component, we evaluate it on these inputs and record its output behavior (i.e. the function return value and any writes to memory). The function signature and IO examples form the component's specification (**SPECIFICATION** in Figure 1).

The next step is to predict which fragments from a population (**FRAGMENT POPULATION**) are most likely to appear in a correct synthesized program, and the structure in which they are most likely to do so in. We use two probabilistic models, IID and Markov to do this (**PREDICTION**). The result of this step is a distribution over program sketches.

The program synthesis phase (**SYNTHESIS**) samples from this distribution to obtain potential program sketches. These are then filled in with instructions to form a candidate program. Candidates are then evaluated on the inputs to see if they match the output. If one does, a correct solution is reported.

2.1 Example

In order to illustrate the workflow illustrated in Figure 1, it is worth considering each step in the context of an example function. We begin with the specification:

```
float f(float *a, float *b, int c) {
  // implementation details are unknown
}
```

The first step is to sample random input values for this type signature, then pass them to the function to observe the output value. For a single example:

```
float a[] = { 0.0, 1.2, -3.4, -5.6 };
float b[] = { -1.0, 1.2, 2.4, 3.2 };
int c = 3;
float ret = f(a, b, c); // ret == -6.72
```

Next, the most likely candidates from the population of sketch fragments are identified. For this problem, a subset of the initial population resembles:

$$\{\ \mathrm{loop}(a), \mathrm{if}(c), \mathrm{loop}(), \ldots, \mathrm{loop}(c), \mathrm{affine}(), \mathrm{index}()\ \}$$

A full description of the semantics of each of these fragment types is given in Section 4. From this population, our IID model identifies the fragments likely to appear in a solution. For this example type signature, these are:

$$\{\ \mathrm{loop}(a, c), \mathrm{loop}(b, c), \mathrm{loop}(a, b, c), \mathrm{linear}()\ \}$$

Next, our Markov model identifies the probability that a particular sequence of these fragments will form the structure of a correct solution:

$$\mathbb{P}(\mathrm{loop}(a, b, c) \circ \mathrm{linear}()) = 0.2$$
$$\ldots$$
$$\mathbb{P}(\mathrm{linear}() \circ \mathrm{loop}(c)) = 0.01$$

Sketches are sampled using these probabilities. For each sketch, code is generated. For the most likely composition above, as C:

```
float f(float *a, float *b, int c) {
  for(int i = 0; i < c; ++i) {
    float ea = a[i], eb = b[i];
  }
}
```

Finally, additional instructions are enumerated and added to the generated code to produce solutions. In the case of this example, a correct program is:

```
float f(float *a, float *b, int c) {
  float g = 0.0f;
  for(int i = 0; i < c; ++i) {
    float d = a[i], e = b[i];
    float h = d * e;
    g = g + h;
  }
  return g;
}
```

By probabilistically identifying likely control flow structures, the synthesis of a complex looping program is reduced to a smaller enumerative search. Sections 4 to 6 give detailed insight into each of these steps individually.

3 Specification

The primary input given to any program synthesizer is a *specification* describing the problem for which to synthesize a solution. While some synthesizers provide specifications in the form of manually constructed examples, PRESYN does not: it specifies synthesis problems in terms of an existing implementation; the goal is to capture the behavior with a synthesized solution.

PRESYN uses only two inputs to specify a synthesis problem: the type signature of the target function, and a shared library containing an implementation with that signature. There are no restrictions on the internal details of this implementation.

3.1 Signature

PRESYN aims for interoperability with C; the programs it synthesizes should be ABI-compatible with the reference implementation. PRESYN supports the primitive C types `char`, `int` and `float`, pointers to these types and `void *`.

This type system allows for greater flexibility in specification than many other synthesizers do. For example, a common restriction is for synthesizers to only consider programs from lists to lists, or lists to single values [11, 24]. In this model, programs are functional and cannot modify state, a restriction not shared by PRESYN. By using an existing, real-world type system, PRESYN expresses problems more naturally than other synthesizers are able to.

3.2 IO Examples

PRESYN generates scalar input parameters by sampling uniformly from fixed-size intervals. Pointer input data is generated by allocating a block of memory and generating each element using the scalar generation strategy. Input data is then passed to the target component; the return value and any pointer parameters make up the recorded output.

We operate under the assumption that observational equivalence (i.e. equivalence over a large set of IO examples) is sufficient. This assumption is shared by other work [10, 19]. While in theory functions may exhibit different behavior on a sparse subset of the input space, we did not find any examples of this in practice.

4 Fragments

PRESYN uses component-based, sketching program synthesis to construct programs; the structure of a solutions is determined by the composition of smaller parts, or *fragments*.

Definition We define a set **F** of fragments, and a set **P** containing concrete programs in a target language, with supported operations:

$$\mathit{compose} : \mathbf{F} \times \mathbf{F} \longrightarrow \mathbf{F}$$
$$\mathit{compile} : \mathbf{F} \longrightarrow \mathbf{P}$$

These operations are total; there are no invalid or intermediate states in this representation. For fragments a, b, c we write:

$$a \circ b \triangleq \mathit{compose}(a, b)$$
$$a \circ b \circ c \triangleq (a \circ b) \circ c$$

Additionally, we consider functions onto **F** (*templates*). For example, the function

$$\mathit{fixed\text{-}loop} : \mathbb{N} \longrightarrow \mathbf{F}$$

represents a fragment template for loops to a fixed upper bound $\in \mathbb{N}$; we use templates to parameterize fragments over argument names in a specification.

Example The semantics of *compose* and *compile* are defined by the implementation of each fragment in **F**. Consider the fragments *fixed-loop*(5) and *skip*. The definition of *compile* for these fragments (to C) is:

$$\mathit{compile}(\mathit{fixed\text{-}loop}(5)) \triangleq \texttt{for(int i=0;i<5;++i) \{\}}$$
$$\mathit{compile}(\mathit{skip}) \triangleq \texttt{\{\}}$$

The *fixed-loop* fragment compiles to an empty loop, while the *skip* compiles to an empty statement. Composition as defined by each fragment produces more interesting structure. For all $f \in \mathbf{F}$:

$$\mathit{compose}(\mathit{fixed\text{-}loop}(5), f) \triangleq \mathit{fixed\text{-}loop}(5)_f$$
$$\mathit{compile}(\mathit{fixed\text{-}loop}(5)_f) \triangleq$$
$$\texttt{for(int i=0;i<5;++i) \{}\mathit{compile}(f)\texttt{\}}$$

4.1 Fragment Population

PRESYN uses a library of 11 different types of fragment. An overview of the most important types of fragment (with C-like pseudocode where appropriate; the use function represents code generated that may use a particular value, ? is possible further composition, and _P is a placeholder value of appropriate type) is as follows:

Linear A basic block into which instructions should later be synthesized; an empty variant prevents instructions.

Fixed Loop Template parameterized on an optional pointer ptr and an integer x:

```
for(int i = 0; i < x; ++i) { ? }
for(int i = 0; i < x; ++i) { use(ptr[i]); ? }
```

Delimiter Loop Template parameterized on a pointer ptr:

```
while(*ptr++ != _P) { use(*ptr); ? }
```

Loop A catch-all for iterations not covered by the two specialized types:

```
while(_P) { ? }
```

If and If-Else Conditional control flow:

```
if(_P) { ? }
if(_P) { ? } else { ? }
```

Seq Execute two fragments, one after the other:

```
? ; ?
```

Affine and Index Synthesize affine and general index expressions respectively, parameterized on ptr. For example:

```
int a_v = ptr[_P * _P + _P]; // e.g.
int v = ptr[_P - _P];        // e.g.
```

The aim for these components was to remain as general as possible so that PRESYN can express many different programs, while not biasing towards one problem domain at the expense of others. We selected some fragments based on intuition for common programming practices (linear, loop, if and if-else), and the remainder from high-level idiomatic patterns used in compiler analyses [26].

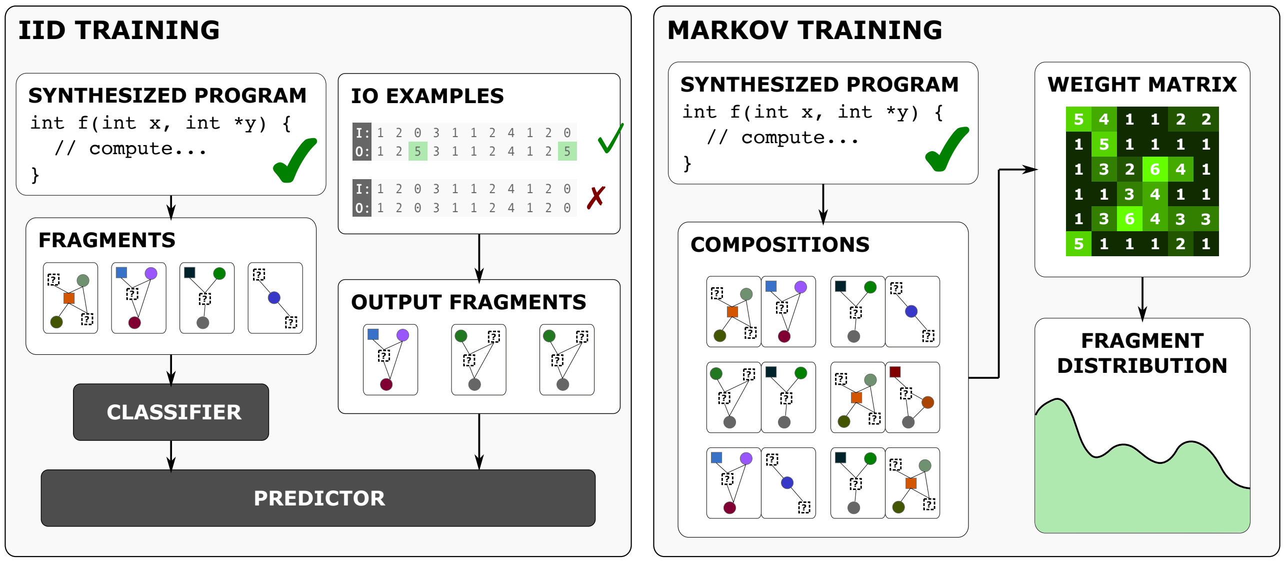
5 Probabilistic Models

PRESYN uses two probabilistic models to guide synthesis. The first, IID, predicts whether each fragment in the population is likely to appear in a solution or not. The second, Markov, creates a probability distribution over sequences of fragments that form compositions.

5.1 IID

After generating input-output examples (**SPECIFICATION** in Figure 1), PRESYN collects an initial set of fragments $\mathbf{F_0} \subseteq \mathbf{F}$ from which solutions may be composed (**PREDICTION** in Figure 1). The library of fragment templates used by PRESYN is too large to perform an exhaustive search, and so accurately *predicting* likely fragments is important.

Suppose $f_0 \circ f_1 \circ \cdots \circ f_n$ is a composition of fragments that can produce a correct solution for a synthesis problem. We aim to predict an initial set of fragments $\mathbf{F_0} \subseteq \mathbf{F}$ that is



```
int f(int x, int *y) {
  // compute...
}
```

```
I: 1 2 0 3 1 1 2 4 1 2 0
O: 1 2 5 3 1 1 2 4 1 2 5

I: 1 2 0 3 1 1 2 4 1 2 0
O: 1 2 0 3 1 1 2 4 1 2 0
```

5	4	1	1	2	2
1	5	1	1	1	1
1	3	2	6	4	1
1	1	3	4	1	1
1	3	6	4	3	3
5	1	1	1	2	1

Figure 2. Training process for our IID and Markov models based on corpus data.

as close to $\{f_0, f_1, \dots f_n\}$ as possible. To do this, we combine two simple ideas:

Fragment Semantics It is possible to make observations about the behavior of the reference implementation based on the input-output examples generated for testing. For example, if a memory region is written to in any of the observed examples, then that region represents an output reference parameter. If such parameters are present, then the set of fragments capable of performing output is collected as $\mathbf{F}_\mathrm{S}$.

Classification Model As well as semantic knowledge, we employ a simple classification model to determine inclusion in $\mathbf{F}_0$ for non-output fragments. To do this, we require a small corpus of training data: type signatures and $\mathbf{F}_0$ for successfully synthesized programs. We train a random forest model, using the type signatures as the input and a binary inclusion indicator for each fragment as the target. This trained model provides a decision function P that determines fragment inclusion in $\mathbf{F}_0$.

Given a trained classifier, we define:

$$\mathbf{F}_0 \triangleq \mathbf{F}_\mathrm{S} \cup \{f \in \mathbf{F} \mid P(f)\}$$

It is clear that it is safe for the prediction of $\mathbf{F}_0$ to over-approximate the true initial set: if additional irrelevant fragments are present, synthesis will simply be slower. However, $\mathbf{F}_0$ may in fact be an *under*-approximation. This is addressed by the next model in the synthesis process, Markov.

5.2 Markov

The set of initial fragments $\mathbf{F}_0$ represents the starting point for synthesis. However, in order to produce a solution structure, fragments must be composed in the correct order; we aim to produce correct compositions $f_0 \circ \cdots \circ f_n$. Additionally, the process of generating these compositions should be robust if $\mathbf{F}_0$ is an under-approximation.

Our model for generating these compositions uses the same training data as IID, but treats fragment occurrences as a Markov model rather than a collection of IID random variables. This model is defined as follows:

Definitions We define auxiliary fragments f_{start}, f_{end} that both act as the identity under composition, along with a function w such that $w(f, f')$ represents the number of occurrences of the composition $f \circ f'$ in the training set.

Training As with IID, training data for Markov comprises correctly synthesized programs together with the composition of fragments used to generate that program. To train the model, each observed composition of fragments has f_{start} prepended and f_{end} appended. The composition sequences are then split into bigrams, and the number of occurrences of each unique bigram is recorded and used to define w.

Generating a Composition Sequences of fragments (representing compositions) can be sampled from Markov as follows. First, we define:

$$s(f) \triangleq \sum_{f' \in \mathbf{F}} w(f, f')$$

Then, starting from f_{start} we sample fragments with conditional probability:

$$\mathbb{P}(f_n | f_{n-1}) = \frac{w(f_{n-1}, f_n)}{s(f_{n-1})}$$

To include the predicted initial set of fragments $\mathbf{F_0}$, we then define an augmented model:

$$w'(f_i, f_j) \triangleq \begin{cases} w(f_i, f_j) \text{ if } f_j \in \mathbf{F_0} \\ 0 \text{ otherwise} \end{cases}$$

$$s'(f) \triangleq \sum_{f' \in \mathbf{F_0}} w'(f, f')$$

Given a parameter $b \in [0, 1]$, fragments are then sampled with conditional probability:

$$\mathbb{P}(f_n | f_{n-1}) = b\frac{w'(f_{n-1}, f_n)}{s'(f_{n-1})} + (1-b)\frac{w(f_{n-1}, f_n)}{s(f_{n-1})}$$

A sequence of fragments $f_0 \circ \cdots \circ f_n$ is generated by sampling until f_{end} is sampled, or to a fixed maximum length. The sequence is then composed to produce a program sketch.

6 Synthesis

The next stage in the synthesis process is to generate concrete programs from candidate sketches generated by our Markov model. They are then executed and tested against the input-output examples collected at the specification step.

Fragments can be compiled to a concrete program in LLVM [35] intermediate representation, but the resulting programs cannot yet be executed:

- They do not perform any meaningful computation; the programs do not yet contain any instructions beyond those used for control flow and program structure.
- The programs contain *placeholders*: typed SSA variables in the program that do not yet have a defined value. For example, a fragment that implements a while-loop may use a boolean placeholder as the loop condition. It cannot be executed until a concrete value is chosen to fill the placeholder.

To resolve these issues, we traverse the dominance tree of the compiled function in-order. For each basic block, we maintain a set of instructions that could be added to it based on the live SSA values at the block's entry. We use an enumerative search to instantiate different concrete programs from the compiled fragment. Once a set of instructions has been selected by this search, a value for each ϕ node and placeholder is chosen by a secondary search.

Safety Because synthesized programs are able to perform potentially arbitrary memory accesses, Presyn implements a conservative, compile-time bounds checking system. During synthesis, all allocated memory passed to the synthesized programs has the same size. If programs make out-of-bounds accesses, the size is increased for future attempts up to a configured maximum. Bounds-checking code is removed when reporting solutions to accurately reflect algorithmic intent.

6.1 Testing

Presyn specifies problems by collecting a large number of input-output examples from the black-box component. The correctness of a candidate program is specified as *observational equivalence*: if the candidate program behaves identically to the reference component on every input generated, then it is correct.

While it is possible that observational equality (even over a large set of examples) is unsound, it is not possible to implement a better decision procedure for an arbitrary black box. This is an observation shared by other work [10].

7 Experimental Setup

Benchmarking program synthesizers against each other fairly while allowing for differences in specification and expression is a challenging problem [30, 41]. This section describes our experimental methodology for evaluating Presyn and comparing its performance to other program synthesizers.

7.1 Overview of Methodology

We identify a collection of benchmark synthesis problems collated from existing work on program synthesis, as well as from real-world software components. For each of these problems, we prepare a specification for Presyn as described in Section 3, then attempt to synthesize a solution.

We evaluated Presyn against five other state-of-the art program synthesizers. Doing so requires benchmarks to be fairly translated; a non-trivial task when accounting for the differences between them.

7.2 Implementations

We had four primary criteria for selecting implementations to compare against. First, they should target a 'general-purpose' language. This ruled out synthesizers such as [18] or [14] that target specific problem domains using SMT-aided formal techniques. Typically, the problem space for programs with arbitrary control flow is too large for these solver-aided techniques [53] and other methods are required. Second, the implementations selected should be representative of the current baseline synthesis performance for target languages similar to their own. Third, the set of implementations should cover a range of different program synthesis methods. Finally, the implementations should be available for evaluation and testing; we aim to make our own implementation and evaluation available in turn. With these goals in mind, the implementations selected are:

SketchAdapt [37] is a neural synthesis approach. It samples programs from a DSL to generate training data for a generative model that predicts programs from IO examples. Its chief innovoation is to adapt to IO complexity, falling back to synthesis when generation is predicted to be too expensive. It represents the state-of-the-art in neural synthesis.

TypeDirect [20] uses program synthesis to enable refactoring of legacy scientific software to use new libraries. It uses a black-box specification to generate IO examples similarly to Presyn, but requires extensive annotation of function signatures to do so.

Makespeare [49] represents the state-of-the-art for genetic program synthesis [12] of programs with complex control flow, and contributes a novel hill-climbing algorithm. Synthesized programs are in a subset of x86 assembly, and problems are specified by providing 'before and after' memory and register state for an abstract machine.

Simpl [53] targets imperative programs written in a small C-like language. Its key innovation is the use of static analysis techniques to prune the search space of candidate programs to ensure that 'dead ends' are not explored unnecessarily. The programs it is evaluated on are designed to model introductory programming exercises with control flow and mutable variables; problems are specified using handwritten input-output examples and a partial program sketch.

λ^2 [25] targets a functional language that permits algebraic data types. It uses a type-aware recursive search process over a space of expressions. Programs synthesized are compositions of a standard library of functions and higher-order functional primitives. Like Simpl, problems are specified by handwritten input-output examples, and in some cases a recursive base case must be supplied.

7.3 Benchmark Collation

We collated a set of 112 synthesis problems from two sources: the benchmarks used by existing synthesizers, and real-world software libraries. The sets of problems selected are summarized in Table 1.

Existing Benchmarks Each synthesizer evaluated uses its own set of synthesis benchmarks, with some partially overlapping. We began with the full set of benchmarks from each of [25, 49, 53] then removed duplicate entries. We then added a representative sample of problems from [37]. To create a more level playing field, we then removed problems for which only one synthesizer was specialized towards. For example, TypeDirect includes extensive type annotations that allow it to synthesize sparse-matrix vector multiplication, and λ^2 includes tree-processing primitive functions.

Black-Box Components In [20], TypeDirect is used to synthesize implementations of 11 functions shared between a set of optimized mathematical libraries such as Intel MKL [2] or Nvidia cuBLAS [5]. We extended this set to include additional mathematical operations found in other real-world libraries: Mathfu [3] and the TI signal processing libraries [1]. Additionally, we identified string processing functions as a common target domain for program synthesis [27, 42, 44]. We identified the C standard library string functions [4] as a realistic target.

7.4 Problem Preparation

The synthesis benchmarks we selected are all specified using different formats. For each problem, we therefore produced a reference implementation in C based on the specifications in the original papers for existing benchmarks, and based on the concrete implementation for real-world code. We then generated an appropriate number of IO examples in the correct format for each one.

In some cases, this required some adaptation. For example, most synthesizers do not support floating point computation; we restated these using integers where appropriate, with the aim of preserving the intent of each problem.

7.5 Synthesizer Help

Although we are interested in black-box IO example synthesis, many of the synthesizers rely on varying degrees of help. We evaluate them with and without the following assistance:

SketchAdapt did not receive any extra help to solve each problem. It does has some difficulties surrounding the integer type, so we convert integers to singleton lists where invalid input types are detected.

TypeDirect TypeDirect requires semantic annotations to be applied to the type signature of a target function (for example, "the pointer `x` points to `N` elements"). These annotations are used as heuristics to guide the search for potential candidate structures.

Makespeare uses a small number of registers values which provide some aid to guiding program generation, however in practise this had little impact on behavior. Unlike other schemes it required a large number of examples, typically in the thousands. We observe that Makespeare is dependent on the large, and varied, set of input examples in many cases.

Simpl relies on a partial program and a list of useful integer constants as input. We provide the correct number of top-level loops and the correct number of variable initializations, which is consistent with many (but not all) of their benchmarks.

λ^2 can exploit an extendible library of base cases. We provide accesses to its standard library of base cases

Presyn does not require additional help.

8 Results

This section presents our experimental results based on the framework presented in the previous section. For cross-evaluation and when comparing IID to Markov, Presyn was trained on a randomly selected subset (15%) of the synthesis problems for which we identified correct fragment structure.

Table 1. Groups of synthesis benchmark problems

Group	N	Description
makespeare	11	Problems that require loops to manipulate arrays of integers in place. We use the full set of benchmarks from [49], modulo those adapted from [53].
simpl-int	15	Arithmetic manipulations of integer values, requiring loops and data-dependent control flow. We use the full set of integer benchmarks from [53].
simpl-array	12	Problems stated over integer arrays, with different styles required (e.g. pairwise iteration, reductions and elementwise computation). We use the full set of array benchmarks from [53].
λ^2	8	Singly-nested integer linked-list manipulation problems from [25], restated for other synthesizers as array problems.
SketchAdapt	10	A series of generated list problems, taken as representative samples from the 500 program evaluation file presented by [37].
string	16	The C standard library's string processing functions [4]. We remove impure functions such as `strtok`.
mathfu	15	Vector-scalar and vector-vector mathematical functions from the Mathfu [3] library.
blas	4	Matrix-vector linear algebra functions from the BLAS [13] standard as synthesized in [20]. We disregard functions such as `spmv` for which extensive assistance was required.
dsp	21	Vector- and matrix-based signal processing functions from the TI signal processing library [1], adapted for platform portability and removing functions with requirements for extensive constant data such as filter taps.

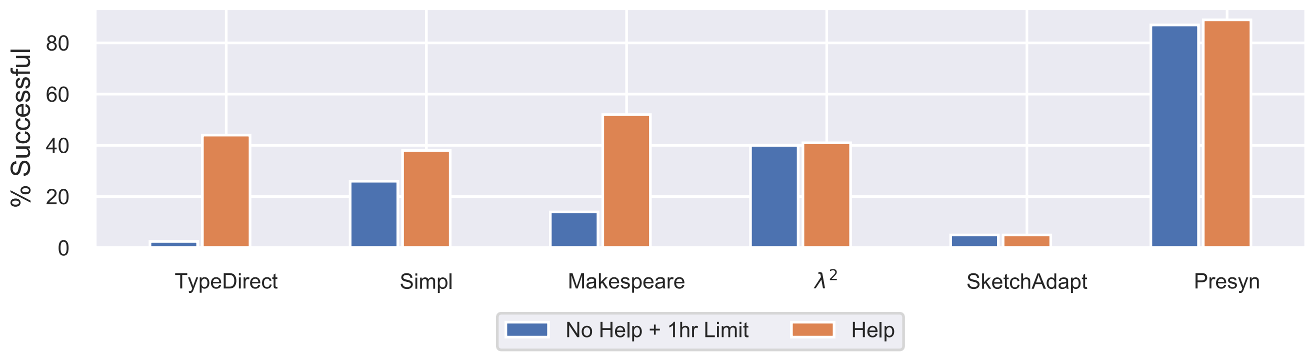


Figure 3. Proportion of the synthesis benchmark set synthesized by each implementation under favorable conditions (see section 7.5), and when restricted by time limits and reduced help.

8.1 Coverage

The primary evaluation criteria for Presyn against other synthesizers is the number of programs it is able to synthesize. We evaluated each synthesizer in three contexts for each synthesis problem: with no additional help beyond IO examples for the problem (i.e. no annotated types or program structure), with the appropriate help as described in Section 7.5 and a conservative timeout, and finally unrestricted (with help). These results are shown in Figure 3.

Presyn is able to successfully synthesize more functions across the set of synthesis benchmarks than each of the other implementations, even when they are given appropriate assistance and unlimited execution time: 89% of the functions evaluated, while the next-best performing (Makespeare) synthesizes only 65%. On real-world code, Presyn synthesizes 93% vs. 63% for TypeDirect. The full results for each synthesizer and library are given in Table 2. Interestingly, it is not the case that each synthesizer performs best on its own benchmarks or that each set of benchmarks is best synthesized with the corresponding implementation; this is likely due to the differences in setup between our experiments and the original work. Nonetheless, it indicates that synthesis is by nature a fragile problem to evaluate experimentally.

Table 2. Proportion of each group of synthesis benchmarks synthesized by each synthesizer under favorable conditions (see section 7.5)

	Group	TypeDirect	Makespeare	Simpl	λ^2	SketchAdapt	Presyn
Benchmarks	makespeare	0.20	0.64	0.09	0.45	0.00	0.55
	simpl-int	0.00	0.80	0.73	0.20	0.00	0.93
	simpl-array	0.58	0.75	0.58	0.58	0.08	0.92
	λ^2	0.43	1.00	0.38	0.75	0.00	1.00
	SketchAdapt	0.20	0.50	0.00	0.10	0.10	0.50
	Mean	**0.26**	**0.73**	**0.39**	**0.39**	**0.04**	**0.79**
Libraries	string	0.00	0.23	0.13	0.56	0.00	0.75
	mathfu	1.00	0.60	0.67	0.47	0.13	1.00
	blas	0.75	0.00	0.00	0.00	0.00	1.00
	dsp	0.90	0.26	0.40	0.38	0.00	1.00
	Mean	**0.92**	**0.33**	**0.38**	**0.48**	**0.04**	**0.93**
	Mean	**0.53**	**0.54**	**0.39**	**0.43**	**0.04**	**0.86**

Makespeare required a large number of examples, typically in the thousands but for some programs, e.g. factorial, thousands of examples would not fit within normal-width integers. Makespeare did not seem particularly influenced by help but requires an extremely long time to synthesize

For Simpl we found that ignoring the partial program input entirely produced very poor results. It struggled significantly on many benchmarks without the desired program structure as input.

For $\boldsymbol{\lambda}^2$, there were a small number of examples where providing a recursive base case for a problem as help made a difference.

SketchAdapt suffered from poor performance on general-purpose problems. We found that SketchAdapt was only successful on trivial examples outside its own evaluation domain and in fact found that it was unable to reproduce its own results once the input values were changed. As it has the least mature implementation, this is perhaps not unexpected. Further, we have taken SketchAdapt further out of its comfort zone than any of the other synthesizers evaluated here—it is intended to perform well on restricted problem domains where the generalizations required are much smaller where the dataset covers a more significant fraction of the program space. Examples of this can be seen within the original paper, where Nye et al. [37] find high accuracy on a number of different subdomains.

When synthesis time is limited or less help is provided for a synthesis problem, Presyn exhibits an even greater advantage over other implementations. Both $\boldsymbol{\lambda}^2$ and Simpl exhibit degraded synthesis when assistance is not given (not shown in Figure 3 is that successful syntheses took up to 300× longer to discover in these cases). Makespeare's use of a genetic algorithm means that it relies on being able to spend a long time searching a space of programs, and struggles when a timeout is imposed.

8.2 Synthesis time and validity

The amount of time spent in synthesis by each scheme varied considerably. Presyn, $\boldsymbol{\lambda}^2$ and Simpl all showed mean synthesis times (for successful cases) of less than 120 s. SketchAdapt required longer, with a mean synthesis time of 914 s. Because of its reliance on genetic search, Makespeare used a mean of 4522 s per synthesized program, with some taking up to 3× this long before being timed out.

The size of programs generated by Presyn varied from 40 to 110 lines of LLVM IR. As we do not have a formal specification, we can only test synthesized programs, not verify them. For every Presyn synthesized program, we automatically generated random and boundary value inputs and checked if outputs matched those from the target blackbox function. In all casess we find them behaviourally equivalent.

Given that we have access to the component code generating the IO examples, we manually inspected the synthesized results. In all cases, using our knowledge, we judged them to be correct. Future work will examine the use of bounded model checking as a means of providing greater assurance.

8.3 Impact of Probabilistic Models

IID Presyn uses two probabilistic models to drive its synthesis. The first, IID, predicts fragments that are likely to form part of a correct solution using a random forest model and a limited model of fragment semantics.

The accuracy of these predictions (measured using the Jaccard coefficient of predicted and true $\mathbf{F_0}$) is given in Table 3. On average, there is an 81% overlap between the predicted value and the true value. IID is not significantly over- or under-approximating in its predictions; this overlap corresponds to < 1 prediction errors per problem.

Table 3. Jaccard coefficient for predictions of the initial fragment set $\mathbf{F}_0$ across each set of synthesis problems.

	Group	**Jaccard**
Benchmarks	makespeare	0.77
	simpl-int	0.85
	simpl-array	0.87
	λ^2	0.72
	SketchAdapt	0.72
Libraries	string	0.72
	mathfu	0.90
	blas	0.95
	dsp	0.77
All	**Mean**	**0.81**

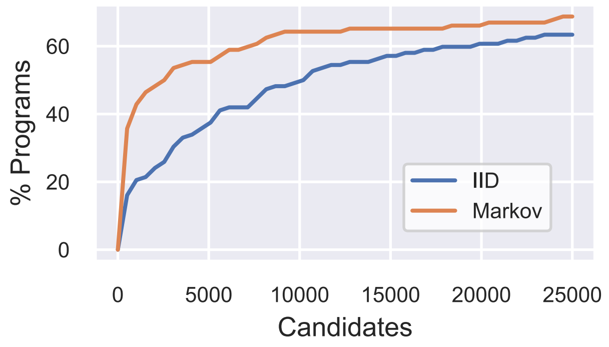


Figure 4. Functions synthesized vs. candidates evaluated using the IID and Markov models of synthesis.

Markov The second model aims to predict the correct order in which to compose fragments to produce a program sketch. To evaluate Markov, we compare the number of programs synthesized by PRESYN against only the IID model. These results are shown in Figure 4. We see that Markov significantly accelerates the synthesis process; it is able to synthesize 60 programs using fewer than half as many candidate programs. In the 'long tail' of programs, the difference is smaller as the complexity of synthesis is dominated by the search for long sequences of instructions.

8.4 Insights into Program Structure

As well as outperforming competing implementations on a wide range of synthesis problems, PRESYN provides interesting statistical insights into the structure of the programs it synthesizes through its use of probabilistic models. The results in this section summarize the distributions learned by PRESYN over its full set of synthesized programs: the models learned on the 15% subset are updated with all subsequent observations, but these are not used during synthesis.

In Figure 5 we show three different insights into the structure of programs synthesized by PRESYN. First, in Figure 5a we see the relative frequency of each type of fragment across benchmark groups. Linear blocks of code are common across all the benchmarks; every program performs some kind of computation. In terms of control flow, the easiest synthesis benchmark suites (simpl-int, simpl-array, mathfu, $\boldsymbol{\lambda}^2$) are those with largely homogeneous control flow across their benchmarks, while the more challenging ones (makespeare, string, SketchAdapt) have far more variation. These results suggest that at least an approximate notion of difficulty for a set of synthesis benchmarks is the heterogeneity in code structure required to solve the problems in that set. Other intuitive structure that can be observed is the ubiquity of nested loops in the blas matrix-vector problems.

Figure 5b shows the number of fragments of each type that appear in synthesized solutions, grouped by unique type signatures (for the 12 most common signatures). Two patterns become apparent from this visualization: the most common type signatures dominate the set of benchmarks, and the fragments used by solutions are generally consistent for each type signature.

Finally, Figure 5c shows the transition probabilities from our Markov model. From this visualization, we can see that the model favors generating smaller programs: most fragments are most likely to be followed by a linear fragment, which is itself likely to end the synthesis.

8.5 Case Studies

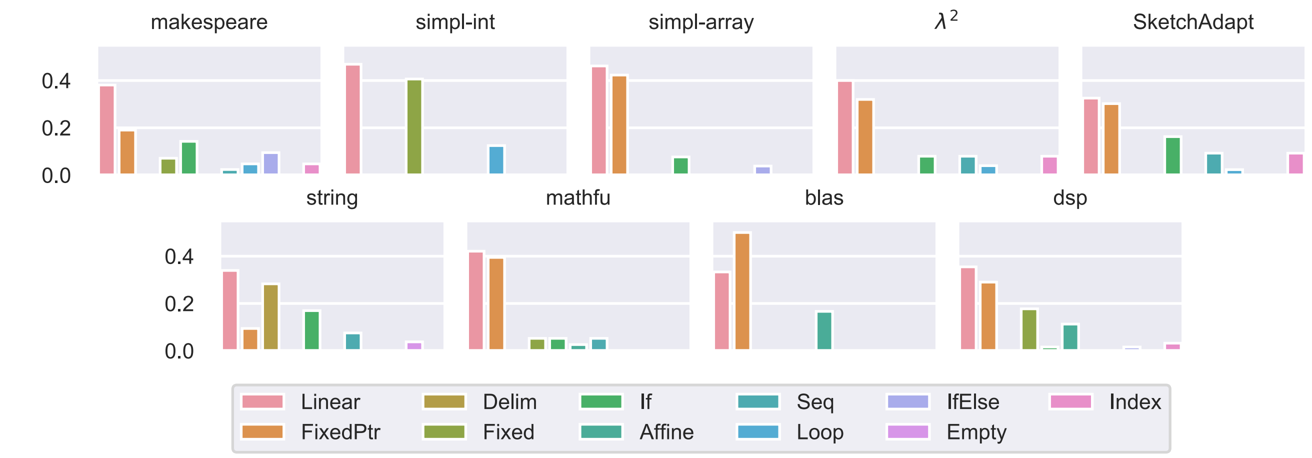
Synthesizing programs that match the behavior of a black box has useful applications in a software engineering context not enabled by other types of synthesis; in this section we detail two of these use cases.

8.5.1 Rejuvenating Legacy Code

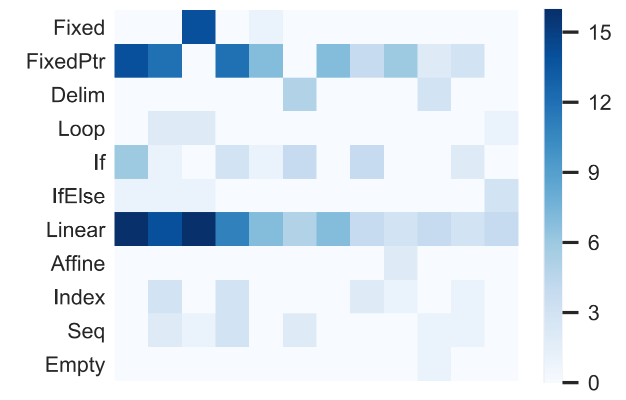
In [20], a system to improve the performance of legacy scientific applications is proposed. It uses TYPEDIRECT to synthesize programs matching the behavior of library functions, such that compatible user code can be remapped to use improved or optimized versions of the functions.

This approach led to considerable performance improvements of up to 10× on deep learning models, and close to 2× on real-world chemical simulation benchmarks. However, the synthesis techniques used are limited: only a small number of functions are considered, and annotations provided by the user provide significant bias to the synthesis process.

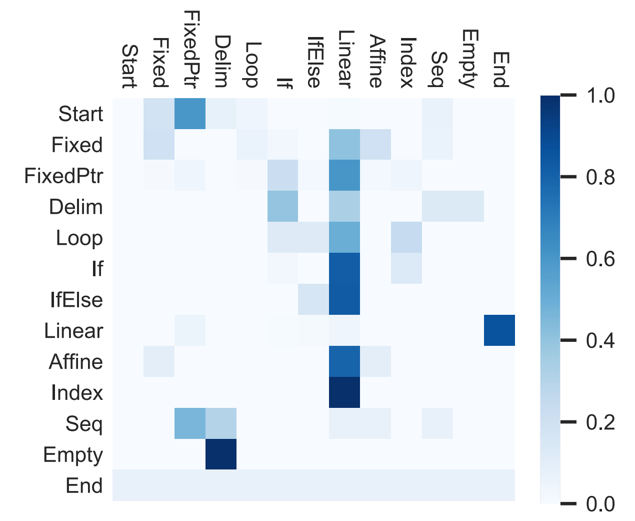
As demonstrated by our results in Section 8, PRESYN outperforms TYPEDIRECT across all the synthesis benchmarks we considered, while requiring no annotations or assistance to be provided by the user for synthesis problems. Of the 11 functions synthesized in [20], PRESYN is able to synthesize 10 (the exception being sparse matrix-vector multiplication, which TYPEDIRECT requires extensive assistance for). Additionally, we were able to synthesize a further 12 functions



(a) Relative frequency of each fragment type in each group of benchmarks.



(b) IID distribution of each fragment type, organized by unique type signatures.



(c) Markov model transition probabilities for each pair of fragment types: from-to is read along rows.

Figure 5. Insights into fragment distributions produced by our IID and Markov models.

from the libraries used in [20] using Presyn; all 22 syntheses were performed without manual type annotation. By using Presyn, more opportunities for performance improvement and an improved API migration can be generated.

8.5.2 Detecting Library Reimplementations

A well-known problem in software engineering is code duplication, particularly when the duplicated functionality has already been implemented by a third-party library [50]. We performed an initial case study to show the application of Presyn to this problem, discovering 163 instances in real-world applications where library functionality is reimplemented.

Table 4 lists the applications we evaluated in this case study; all were written in C or C++. Our methodology was as follows: first, function implementations from the libraries in Table 1 were synthesized using Presyn. Then, an existing tool [20] was used to convert each of the synthesized functions into a set of SMT constraints, which were then passed to an off-the-shelf solver [26] to discover satisfying instances in each of the applications. The number discovered in each application is given in Table 4.

While the conversion of a program to a constraint description and the search for those constraints are due to prior work [20, 26], our application of program synthesis to discover code in user applications that could be refactored to use a library is novel. This integration with existing compiler tools is possible because of the novel design of Presyn; such integrations are not possible with other implementations.

Table 4. Number of discovered instances in user applications where code duplicates library functionality.

Software	LoC	Library	N
FFmpeg	1,061,655	Mathfu	10
		BLAS	3
		DSP	11
Coreutils	66,355	String	10
GraphicsGems	46,619	Mathfu	35
		DSP	26
Darknet	21,299	BLAS	5
		Mathfu	3
		DSP	5
Nanvix	11,226	String	10
ETR	2,399	Mathfu	29
		DSP	16
Total			**163**

9 Related Work

Sketching One of the most important developments in program synthes is the idea of sketching [54, 55]; it has been used for a wide variety of purposes including auto-parallelization [23] and SQL query generation [57]. Recent technical developments include recursive tree transformations [31] and improved modularity of sketches [52]. Generally, such schemes require external sketch suggestions: Presyn automates this by constructing priors over program corpora.

Types Annotated types signatures or hints are often used to direct program synthesis, most commonly for functional programs [38, 39]. Myth [40] uses type signatures alongside examples to synthesize recursive functional programs, while [46] uses refinement types to guide the search process [46]. In [21] extended type information is suggested as a means of improving program synthesis, and in [20] a similar approach is used as a means of accessing heterogeneous accelerators for scientific applications. Our work considers a wider, more diverse class of libraries and applications and does not require human annotations or hints.

Neural Program Synthesis The machine learning community has long studied programming by example and input-output based program synthesis. Recent work has examined both induction (with a learned latent version of the program) and generation, which uses a language model to generate programs [9, 22, 43]

A recent trend in program synthesis is the use of neural networks and machine learning. Generative approaches are focused on developing neural architectures that interpret programs intrinsically [48]. These approaches so far struggle to generalize to large problem sizes and to complex semantic models such as the compiler IR used by Presyn.

Others have used neural components to improve the performance of an existing synthesizer. For example, both [11] and [58] aim to learn from input-output examples; both require fixed-size inputs and outputs and use a small DSL to generate training examples. Learned programs are limited to list processing tasks; the DSLs targeted by these (and similar implementations such as SketchAdapt [37]) also rely on high level primitive including (for example) primitives to tokenize strings or perform list manipulations.

White-Box Our approach to synthesis (from *black box* oracles) is less widely studied than the corresponding *white box* problem, where the internal structure or implementation of a reference oracle is known. Existing programs in specific problem domains are often used to synthesize optimized implementations in a domain-specific language: the Halide image-processing language [47] is targeted by several oracle-based approaches, based on x86 assembly [36], Fortran [34] and C++ [8] respectively. In [15], abstract descriptions of software functionality are learned based on dynamic traces and source code in order to facilitate refactoring and other analyses.

Operating under the assumption of a black-box oracle means that many existing approaches in program synthesis do not apply or fail to generalize to our context [16, 17]. By using a black-box oracle we are able to avoid issues of generalization across datasets [33, 42].

Synthesizing Imperative Programs Prior work in imperative synthesis frequently focuses on straight-line code [29, 51] or has to make special provision for control-flow [27]. Simpl[53] overcomes this problem by assuming a partial program is already provided (such as a loop structure).

10 Conclusion

In this paper we have addressed the novel problem of *black box* program synthesis, where problem specifications are based on the observed behavior of an existing component (rather than a human description). Our synthesizer, Presyn, achieves better performance across a wide range of synthesis benchmarks (composed of both new and existing problems) than five other competing synthesizers.

As well as strong synthesis performance, the simple probabilistic models used to implement Presyn provide interesting insights into the structure and difficulty of synthesis problems. Additionally, we demonstrate two case studies where Presyn can be used to solve software engineering problems. Our results in these case studies are promising and we hope to perform further investigation in this direction in the future.

References

[1] [n. d.]. C674x DSPLIB - Texas Instruments Wiki. http://processors.wiki.ti.com/index.php/C674x_DSPLIB.

[2] [n. d.]. Intel® Math Kernel Library (MKL). https://software.intel.com/mkl.

[3] [n. d.]. Mathfu. https://github.com/google/mathfu.

[4] [n. d.]. Null-terminated byte strings. https://en.cppreference.com/w/c/string/byte.

[5] [n. d.]. NVIDIA cuBLAS. https://developer.nvidia.com/cublas.

[6] 2020. witheld for blind review. (2020).

[7] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. 2018. Counterexample guided inductive synthesis modulo theories. In *International Conference on Computer Aided Verification*. Springer, 270–288.

[8] Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. 2019. Automatically Translating Image Processing Libraries to Halide. *ACM Trans. Graph.* 38, 6 (Nov. 2019), 204:1–204:13. https://doi.org/10.1145/3355089.3356549

[9] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 81.

[10] Shengwei An, Sasa Misailovic, Roopsha Samanta, and Rishabh Singh. 2019. Augmented Example-Based Synthesis.

[11] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. DeepCoder: Learning to Write Programs. (Nov. 2016). https://arxiv.org/abs/1611.01989v2

[12] Wolfgang Banzhaf, Frank D. Francone, Robert E. Keller, and Peter Nordin. 1998. *Genetic Programming: An Introduction: On the Automatic Evolution of Computer Programs and Its Applications.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[13] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. 2001. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Software* 28 (2001), 135–151.

[14] Eric Butler, Emina Torlak, and Zoran Popović. 2017. Synthesizing Interpretable Strategies for Solving Puzzle Games. In *Proceedings of the 12th International Conference on the Foundations of Digital Games (FDG '17)*. ACM, New York, NY, USA, 10:1–10:10. https://doi.org/10.1145/3102071.3102084

[15] José P. Cambronero, Thurston H. Y. Dang, Nikos Vasilakis, Jiasi Shen, Jerry Wu, and Martin C. Rinard. 2019. Active Learning for Software Engineering. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2019)*. ACM, New York, NY, USA, 62–78. https://doi.org/10.1145/3359591.3359732

[16] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2019. Multi-Modal Synthesis of Regular Expressions. (Aug. 2019).

[17] Yanju Chen, Ruben Martins, and Yu Feng. 2019. Maximal Multi-Layer Specification Synthesis. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. ACM, New York, NY, USA, 602–612. https://doi.org/10.1145/3338906.3338951

[18] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*.

[19] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Progr ams. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Funct ional Programming (ICFP '00)*. ACM, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

[20] B. Collie, P. Ginsbach, and M. F. P. O'Boyle. 2019. Type-Directed Program Synthesis and Constraint Generation for Library Portability. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 55–67. https://doi.org/10.1109/PACT.2019.00013

[21] Bruce Collie and Michael O'Boyle. 2019. Augmenting Type Signatures for Program Synthesis. *arXiv:1907.05649 [cs]* (July 2019). arXiv:cs/1907.05649 http://arxiv.org/abs/1907.05649

[22] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. Synthesizing benchmarks for predictive modeling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 86–99.

[23] Grigory Fedyukovich, Maaz Bin Safeer Ahmad, and Rastislav Bodik. 2017. Gradual Synthesis for Static Parallelization of Single-pass Array-processing Programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 572–585. https://doi.org/10.1145/3062341.3062382

[24] 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)*. ACM, New York, NY, USA, 420–435. https://doi.org/10.1145/3192366.3192382

[25] 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 (PLDI '15)*. ACM, New York, NY, USA, 229–239. https://doi.org/10.1145/2737924.2737977

[26] Philip Ginsbach, Toomas Remmelg, Michel Steuwer, Bruno Bodin, Christophe Dubach, and Michael F. P. O'Boyle. 2018. Automatic Matching of Legacy Code to Heterogeneous APIs: An Idiomatic Approach. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 139–153. https://doi.org/10.1145/3173162.3173182

[27] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

[28] Sumit Gulwani. 2016. Programming by Examples (and its Applications in Data Wrangling). In *Verification and Synthesis of Correct and Secure Systems*. IOS Press. https://www.microsoft.com/en-us/research/publication/programming-examples-applications-data-wrangling/

[29] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-Free Programs. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 62–73. https://doi.org/10.1145/1993498.1993506

[30] Thomas Helmuth and Lee Spector. 2015. General Program Synthesis Benchmark Suite. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation (GECCO '15)*. ACM, New York, NY, USA, 1039–1046. https://doi.org/10.1145/2739480.2754769

[31] Jeevana Priya Inala, Nadia Polikarpova, Xiaokang Qiu, Benjamin S. Lerner, and Armando Solar-Lezama. 2017. Synthesis of Recursive ADT Transformations from Reusable Templates. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science)*, Axel Legay and Tiziana Margaria (Eds.). Springer Berlin Heidelberg, 247–263.

[32] Susmit Jha and Sanjit A Seshia. 2017. A theory of formal synthesis via inductive learning. *Acta Informatica* 54, 7 (2017), 693–726.

[33] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. 2018. Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples. *arXiv:1804.01186 [cs]* (Sept. 2018). arXiv:cs/1804.01186

[34] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified Lifting of Stencil Computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 711–726. https://doi.org/10.1145/2908080.2908117

[35] Chris Lattner. 2002. *LLVM: An Infrastructure for Multi-Stage Optimization*. Ph.D. Dissertation. Computer Science Dept., University of Illinois at Urbana-Champaign.

[36] Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, and Saman Amarasinghe. 2015. Helium: Lifting High-Performance Stencil Kernels from Stripped X86 Binaries to Halide DSL Code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 391–402. https://doi.org/10.1145/2737924.2737974

[37] Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. 2019. Learning to Infer Program Sketches. In *International Conference on Machine Learning*. 4861–4870.

[38] Peter-Michael Osera. 2019. Constraint-Based Type-Directed Program Synthesis. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development (TyDe 2019)*. ACM, New York, NY, USA, 64–76. https://doi.org/10.1145/3331554.3342608

[39] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-Example-Directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 619–630. https://doi.org/10.1145/2737924.2738007

[40] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-Example-Directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 619–630. https://doi.org/10.1145/2737924.2738007

[41] Edward Pantridge, Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. 2017. On the Difficulty of Benchmarking Inductive Program Synthesis Methods. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '17)*. ACM, New York, NY, USA, 1589–1596. https://doi.org/10.1145/3067695.3082533

[42] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2016. Neuro-Symbolic Program Synthesis. *arXiv:1611.01855 [cs]* (Nov. 2016). arXiv:cs/1611.01855

[43] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2016. Neuro-Symbolic Program Synthesis. (Nov. 2016). https://arxiv.org/abs/1611.01855v1

[44] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. 2014. Test-Driven Synthesis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 408–418. https://doi.org/10.1145/2594291.2594297

[45] Elizabeth Polgreen, Ralph Abboud, and Daniel Kroening. 2020. CounterExample Guided Neural Synthesis. *arXiv preprint arXiv:2001.09245* (2020).

[46] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 522–538. https://doi.org/10.1145/2908080.2908093

[47] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[48] Scott Reed and Nando de Freitas. 2015. Neural Programmer-Interpreters. (Nov. 2015). https://arxiv.org/abs/1511.06279v4

[49] Christopher D. Rosin. 2018. Stepping Stones to Inductive Synthesis of Low-Level Looping Programs. *arXiv:1811.10665 [cs]* (Nov. 2018). arXiv:cs/1811.10665

[50] C. K. Roy, M. F. Zibran, and R. Koschke. 2014. The Vision of Software Clone Management: Past, Present, and Future (Keynote Paper). In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. 18–33. https://doi.org/10.1109/CSMR-WCRE.2014.6747168

[51] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. 2017. Souper: A Synthesizing Superoptimizer. *arXiv:1711.04422 [cs]* (Nov. 2017). arXiv:cs/1711.04422 http://arxiv.org/abs/1711.04422

[52] Rohit Singh, Rishabh Singh, Zhilei Xu, Rebecca Krosnick, and Armando Solar-Lezama. 2014. Modular Synthesis of Sketches Using Models. In *Verification, Model Checking, and Abstract Interpretation (Lecture Notes in Computer Science)*, Kenneth L. McMillan and Xavier Rival (Eds.). Springer Berlin Heidelberg, 395–414.

[53] Sunbeom So and Hakjoo Oh. 2017. Synthesizing Imperative Programs from Examples Guided by Static Analysis. In *Static Analysis (Lecture Notes in Computer Science)*, Francesco Ranzato (Ed.). Springer International Publishing, Freiburg, Germany, 364–381.

[54] Armando Solar-Lezama. 2009. The Sketching Approach to Program Synthesis. In *Programming Languages and Systems (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 4–13. https://doi.org/10.1007/978-3-642-10672-9_3

[55] Armando Solar-Lezama. 2013. Program sketching. *International Journal on Software Tools for Technology Transfer* 15, 5-6 (2013), 475–495.

[56] C. Teyton, J. Falleri, and X. Blanc. 2013. Automatic Discovery of Function Mappings between Similar Libraries. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. 192–201. https://doi.org/10.1109/WCRE.2013.6671294

[57] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. 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)*. ACM, New York, NY, USA, 452–466. https://doi.org/10.1145/3062341.3062365

[58] Amit Zohar and Lior Wolf. 2018. Automatic Program Synthesis of Long Programs with a Learned Garbage Collector. In *Proceedings of the 32Nd International Conference on Neural Information Processing Systems (NIPS'18)*. Curran Associates Inc., USA, 2098–2107.