Synthesizing Imperative Programs from Examples Guided by Static Analysis

Sunbeom So and Hakjoo Oh

Korea University

Abstract. We present a novel algorithm for efficiently synthesizing imperative programs from examples. Given a set of input-output examples and a partial program, our algorithm generates a complete program that is consistent with every example. Our algorithm is based on enumerative synthesis, which explores all candidate programs in increasing size until it finds a solution. This algorithm, however, is too slow to be used in practice. Our key idea to accelerate the speed is to perform static analysis alongside the enumerative search, in order to "statically" identify and safely prune out partial programs that eventually fail to be a solution. We have implemented our algorithm in a tool, SIMPL, and evaluated it on 30 introductory programming problems gathered from online forums. The results show that our static analysis approach improves the speed of enumerative synthesis by 25x on average.

1 Introduction

In this paper, we show that semantic-based static analysis (à la abstract interpretation) can be effectively used to speed up enumerative program synthesis. While static analysis has played key roles in program bug-finding, verification, and optimization, its application to program synthesis remains to be seen. Static type systems have been used for synthesizing functional programs [8, 9, 24, 26], but type-directed synthesis is not generally applicable to languages with, for instance, dynamic or unsafe type systems. This paper explores an alternative, static-analysis-guided program synthesis.

We focus on the problem of synthesizing imperative programs, where type-based techniques are not useful. The inputs of our algorithm are a partial program with constraints on variables and constants, and input-output examples that specify a resulting program's behavior. The output is a complete program whose behavior matches all of the given input-output examples.

The key novelty of our algorithm is to combine enumerative program synthesis and static analysis. It basically enumerates every possible candidate program in increasing size until it finds a solution. This algorithm, however, is too slow to be used due to the huge search space of programs. Our key idea to accelerate the speed is to perform static analysis, in order to "statically" identify and safely prune out partial programs that eventually fail to be a solution. More specifically, we prune partial programs whose over-approximated results do not satisfy expected behaviors defined by input-output examples. We formalize our pruning technique and its safety property.

The experimental results show that our static-analysis-guided algorithm is remarkably effective to synthesize imperative programs. We have implemented the algorithm

in a tool, SIMPL, and evaluated its performance on 30 introductory programming tasks manipulating integers and arrays of integers. The benchmarks are gathered from online forums and include problems non-trivial for beginner-level programmers. With our pruning technique, SIMPL is fast enough to solve each problem in 6.6 seconds on average. Without it, however, the baseline algorithm, which already adopts well-known optimization techniques, takes 165.5 seconds (25x slowdown) on average.

Contributions We summarize our contributions below:

- We present a new algorithm for synthesizing imperative programs from examples. The key idea is to combine enumerative program synthesis with static analysis, which greatly accelerates the speed while guaranteeing to find a solution.
- We prove the effectiveness of our algorithm on 30 introductory programming problems gathered from online forums, including non-trivial ones for beginner-level students. The results show that our algorithm quickly solves the problems, in 6.6 seconds on average.
- We make our tool, SIMPL, and benchmark problems publicly available, so that readers can use our tool and reproduce the experimental results:

`http://prl.korea.ac.kr/simpl`

2 Motivating Examples

In this section, we showcase SIMPL with four programming problems. To use SIMPL, users need to provide (1) a set of input-output examples, (2) a partial program, and (3) resources that SIMPL can use. The resources consist of a set of integers, a set of integer-type variables, and a set of array-type variables. The job of SIMPL is to complete the partial program w.r.t. the input-output examples, using only the given resources.

Problem 1 (Reversing integer) Consider the problem of writing the function `reverse` that reverses a given natural number. For example, given 12, the function should return 21. Suppose a partial program is given as

```
reverse (n) { r := 0; while (?) { ? }; return r; }
```

where **?** denotes holes that need to be completed. Suppose further SIMPL is provided with input-output examples $\{1 \mapsto 1, 12 \mapsto 21, 123 \mapsto 321\}$, integers $\{0, 1, 10\}$, and integer variables $\{\mathtt{n}, \mathtt{r}, \mathtt{x}\}$, where $a \mapsto b$ indicates a single example with input a and output b.

Given these components, SIMPL produces the solution in Figure 1(a) in 2.5 seconds. Note that SIMPL finds out the integer '1' is unnecessary and the final program does not contain it.

Problem 2 (Counting) The next problem is to write a function that counts the number of each digit in an integer. The program takes an integer and an array as inputs, where each element of the array is initially 0. As output, the program returns that array but now each array element at index i stores the number of is that occur in the given integer. For

```
reverse(n){
  r := 0;
  while (n > 0){
    x := n % 10;
    r := r * 10;
    r := r + x;
    n := n / 10;
  };
  return r;
}
```

(a) Problem1 (# 16)

```
count(n,a){

  while (n > 0){
    t := n % 10;
    a[t] := a[t] + 1;
    n := n / 10;
  };
  return a;

}
```

(b) Problem 2 (# 30)

```
sum(n){
  r := 0;
  while (n > 0){
    t := n;
    while (t > 0){
      r := r + t;
      t := t - 1;
    };
    n := n - 1;
  };
  return r;
}
```

(c) Problem 3 (# 14)

```
abssum(a, len){
  r := 0;
  i := 0;
  while (i < len){
    if (a[i] < 0)
     {r := r - a[i];}
    else
     {r := r + a[i];}
    i := i + 1;
  };
  return r;
}
```

(d) Problem 4 (# 29)

Fig. 1. Synthesized results by SIMPL (in the boxes). #n denotes the number in Table 1.

example, when a tuple $(220, \langle 0,0,0\rangle)$ is given, the function should output $\langle 1,0,2\rangle$; 0 occurs once, 1 does not occur, and 2 occurs twice in '220'. Suppose the partial program is given as

```
count (n,a) { while(?){?}; return a;}
```

with examples $\{(11, \langle 0,0\rangle) \mapsto \langle 0,2\rangle, (220, \langle 0,0,0\rangle) \mapsto \langle 1,0,2\rangle\}$, integers $\{0,1,10\}$, integer variables $\{\mathtt{i},\mathtt{n},\mathtt{t}\}$, and an array variable $\{\mathtt{a}\}$.

For this problem, SIMPL produces the program in Figure 1(b) in 0.2 seconds. Note that `i` is not used though it is given as a usable resource.

Problem 3 (Sum of sum) The third problem is to compute $1 + (1+2) + ... + (1+2+...+n)$ for a given integer n. Suppose the partial program

```
sum(n){ r := 0; while(?){?}; return r;}
```

is given with examples $\{1 \mapsto 1, 2 \mapsto 4, 3 \mapsto 10, 4 \mapsto 20\}$, integers $\{0,1\}$, and integer-type variables $\{\mathtt{n},\mathtt{t},\mathtt{r}\}$.

Then, SIMPL produces the program in Figure 1(c) in 37.6 seconds. Note that SIMPL newly introduced the inner loop, which is absent in the partial program.

Problem 4 (Absolute sum) The last problem is to sum the absolute values of all the elements in a given array. We provide the partial program:

```
absum(a, len){ r := 0; i := 0;
  while(i < len){ if(?){?} else{?}; i:=i+1;};
return r;}
```

where the goal is to complete the condition and bodies of the if-then-else statement. Additionally, given a set of input-output examples $\{(\langle -1, -2\rangle, 2) \mapsto 3, (\langle 2, 3, -4\rangle, 3) \mapsto 9\}$, an integer $\{0\}$, integer variables $\{\texttt{r}, \texttt{i}\}$, and an array variable $\{\texttt{a}\}$, SIMPL produces the program in Figure 1(d) in 12.1 seconds.

Finally, we emphasize the following points regarding usage scenarios of SIMPL:

- SIMPL requires only a few input-output examples. In experiments (Table 1), it was enough to provide 2–4 input-output examples for each programming task. These examples are simple enough to conceive.
- Resources can be over-estimated if uncertain (Problem1, Problem2). The unnecessary resources will be ignored by SIMPL.

3 Problem Definition

Language We designed an imperative language that is small yet expressive enough to deal with various introductory programming problems. The syntax of the language is defined by the grammar in Figure 2.

A l-value (l) is a variable (x) or an array reference ($x[y]$). An arithmetic expression (a) is an integer constant (n), an l-value (l), or a binary operation ($\oplus$). A boolean expression (b) is a boolean constant ($true$, $false$), a binary relation ($\prec$), a negation ($\neg$), or a logical conjunction ($\wedge$) and disjunction ($\vee$). Commands include assignment ($l := a$), skip ($skip$), sequence ($c_1; c_2$), conditional statement ($if\ b\ c_1\ c_2$), and while-loop ($while\ b\ c$).

A program $P = (x, c, y)$ is a command with input and output variables, where x is the input variable, c is the command, and y is the output variable. The input and output variables x and y can be either of integer or array types. For presentation brevity, we assume that the program takes a single input, but our implementation supports multiple input variables as well.

An unusual feature of the language is that it allows to write incomplete programs. Whenever uncertain, any arithmetic expressions, boolean expressions, and commands can be left out with holes ($\Diamond, \triangle, \square$). The goal of our synthesis algorithm is to automatically complete such partial programs.

$$
\begin{array}{l}
\oplus \ ::= \ + \mid - \mid * \mid / \mid \%, \quad \prec \ ::= \ \ = \mid > \mid < \\
l \ ::= \ x \mid x[y], \quad a \ ::= \ n \mid l \mid l_1 \oplus l_2 \mid l \oplus n \mid \ \Diamond \\
b \ ::= \ true \mid false \mid l_1 \prec l_2 \mid l \prec n \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid \neg b \mid \ \triangle \\
c \ ::= \ l := a \mid skip \mid c_1; c_2 \mid if\ b\ c_1\ c_2 \mid while\ b\ c \mid \ \square
\end{array}
$$

Fig. 2. Language

$$
\begin{aligned}
\mathcal{A}[\![n]\!](m) &= n \\
\mathcal{A}[\![x]\!](m) &= m(x) \\
\mathcal{A}[\![x[y]]\!](m) &= m(x)_{m(y)} \\
\mathcal{A}[\![l_1 \oplus l_2]\!](m) &= \mathcal{A}[\![l_1]\!](m) \oplus \mathcal{A}[\![l_2]\!](m) \\
\mathcal{A}[\![l \oplus n]\!](m) &= \mathcal{A}[\![l]\!](m) \oplus n \\
\\
\mathcal{B}[\![true]\!](m) &= true \\
\mathcal{B}[\![false]\!](m) &= false \\
\mathcal{B}[\![l_1 \prec l_2]\!](m) &= \mathcal{A}[\![l_1]\!](m) \prec \mathcal{A}[\![l_2]\!](m) \\
\mathcal{B}[\![l \prec n]\!](m) &= \mathcal{A}[\![l]\!](m) \prec n \\
\mathcal{B}[\![b_1 \wedge b_2]\!](m) &= \mathcal{B}[\![b_1]\!](m) \wedge \mathcal{B}[\![b_2]\!](m) \\
\mathcal{B}[\![b_1 \vee b_2]\!](m) &= \mathcal{B}[\![b_1]\!](m) \vee \mathcal{B}[\![b_2]\!](m) \\
\mathcal{B}[\![\neg b]\!](m) &= \neg \mathcal{B}[\![b]\!](m)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}[\![x := a]\!](m) &= m[x \mapsto \mathcal{A}[\![a]\!](m)] \\
\mathcal{C}[\![x[y] := a]\!](m) &= m[x \mapsto m(x)^{\mathcal{A}[\![a]\!](m)}_{m(y)}] \\
\mathcal{C}[\![skip]\!](m) &= m \\
\mathcal{C}[\![c_1; c_2]\!](m) &= (\mathcal{C}[\![c_2]\!] \circ \mathcal{C}[\![c_1]\!])(m) \\
\mathcal{C}[\![if\ b\ c_1\ c_2]\!](m) &= \mathsf{cond}(\mathcal{B}[\![b]\!], \mathcal{C}[\![c_1]\!], \mathcal{C}[\![c_2]\!])(m) \\
\mathcal{C}[\![while\ b\ c]\!](m) &= (\mathsf{fix}\ F)(m) \\
\text{where } F(g) &= \mathsf{cond}(\mathcal{B}[\![b]\!], g \circ \mathcal{C}[\![c]\!], \lambda x.x) \\
\mathsf{cond}(p, f, g)(m) &= \begin{cases} f(m) \text{ if } p(m) = true \\ g(m) \text{ if } p(m) = false \end{cases}
\end{aligned}
$$

Fig. 3. Semantics of the language

The semantics of the language is defined for programs without holes. Let $\mathbb{X}$ be the set of program variables, which is partitioned into integer and array types, i.e., $\mathbb{X} = \mathbb{X}_i \uplus \mathbb{X}_a$. A memory state

$$m \in \mathbb{M} = \mathbb{X} \to \mathbb{V}, \quad v \in \mathbb{V} = \mathbb{Z} + \mathbb{Z}^*$$

is a partial function from variables to values ($\mathbb{V}$). A value is either an integer or an array of integers. An array $a \in \mathbb{Z}^*$ is a sequence of integers. For instance, we write $\langle 1, 2, 3 \rangle$ for the array of integers 1, 2, and 3. We write $|a|$, a_i, and a_i^k for the length of a, the element at index i, and the array $a_0 \ldots a_{i-1} k a_{i+1} \ldots a_{|a|-1}$, respectively.

The semantics of the language is defined by the functions:

$$\mathcal{A}[\![a]\!] : \mathbb{M} \to \mathbb{V}, \quad \mathcal{B}[\![b]\!] : \mathbb{M} \to \mathbb{B}, \quad \mathcal{C}[\![c]\!] : \mathbb{M} \to \mathbb{M}$$

where $\mathcal{A}[\![a]\!]$, $\mathcal{B}[\![b]\!]$, and $\mathcal{C}[\![c]\!]$ denote the semantics of arithmetic expressions, boolean expressions, and commands, respectively. Figure 3 presents the denotational semantics, where fix is a fixed point operator. Note that the semantics for holes is undefined.

Synthesis Problem A synthesis task is defined by the five components:

$$((x, c_0, y), \mathcal{E}, \Gamma, \mathbb{X}_i, \mathbb{X}_a)$$

where (x, c_0, y) is an initial incomplete program with holes, and $\mathcal{E} \subseteq \mathbb{V} \times \mathbb{V}$ is a set of input-output examples. The resource components to be used are given as a triplet, where Γ is a set of integers, $\mathbb{X}_i$ is a set of integer-type variables, and $\mathbb{X}_a$ is a set of array-type variables.[1] The goal of our synthesis algorithm is to produce a complete command c without holes such that

- c uses constants and variables in Γ and $\mathbb{X}_i \cup \mathbb{X}_a$, and
- c is consistent with every input-output example:

$$\forall (v_i, v_o) \in \mathcal{E}.\ \big(\mathcal{C}[\![c]\!]([x \mapsto v_i])\big)(y) = v_o.$$

[1] Resource variables may not exactly coincide with program variables (e.g., Problem 2 in Section 2). However, for legibility, we abuse the notation between them.

4 Synthesis Algorithm

In this section, we present our synthesis algorithm that combines enumerative search with static analysis. We formalize the synthesis problem as a state search problem (Section 4.1) and presents a basic enumerative search algorithm (Section 4.2). Section 4.3 presents our pruning with static analysis.

4.1 Synthesis as State-Search

We first reduce the synthesis task to a state-search problem. Consider a synthesis task $((x, c_0, y), \mathcal{E}, \Gamma, \mathbb{X}_i, \mathbb{X}_a)$. The corresponding search problem is defined by the transition system

$$(S, \rightsquigarrow, s_0, F)$$

where S is a set of states, $(\rightsquigarrow) \subseteq S \times S$ is a transition relation, $s_0 \in S$ is an initial state, and $F \subseteq S$ is a set of solution states.

- **States** : A state $s \in S$ is a command possibly with holes, which is defined by the grammar in Section 3.
- **Initial state** : An initial state s_0 is an initial partial command c_0.
- **Transition relation** : Transition relation $(\rightsquigarrow) \subseteq S \times S$ determines a next state that is immediately reachable from a current state. The relation is defined as a set of inference rules in Figure 4. Intuitively, a hole can be replaced by an arbitrary expression (or command) of the same type. Given a state s, we write $\mathsf{next}(s)$ for the set of all immediate next states from s, i.e., $\mathsf{next}(s) = \{s' \mid s \rightsquigarrow s'\}$.

  Example 1. Given $\Gamma = \{1\}$, $\mathbb{X}_i = \{x\}$ and $\mathbb{X}_a = \emptyset$, consider a state $s = (\square; r := 1; r := \Diamond)$. Then, $\mathsf{next}(s) = \{(x := \Diamond; r := 1; r := \Diamond), (\mathit{skip}; r := 1; r := \Diamond), (\square; \square; r := 1; r := \Diamond), (\mathit{if}\ \triangle\ \square\ \square; r := 1; r := \Diamond), (\mathit{while}\ \triangle\ \square; r := 1; r := \Diamond), (\square; r := 1; r := 1;), (\square; r := 1; r := x), (\square; r := 1; r := x + x), (\square; r := 1; r := x - x), (\square; r := 1; r := x * x), (\square; r := 1; r := x/x), (\square; r := 1; r := x\%x), (\square; r := 1; r := x + 1), (\square; r := 1; r := x - 1), (\square; r := 1; r := x * 1), (\square; r := 1; r := x/1), (\square; r := 1; r := x\%1)\}$.

  We write $s \not\rightsquigarrow$ for terminal states, i.e., states with no holes.
- **Solution states** : A state s is a solution iff s is a terminal state and it is consistent with all input-output examples:

$$\mathsf{solution}(s) \iff s \not\rightsquigarrow \ \wedge \forall (v_i, v_o) \in \mathcal{E}.\ \big(\mathcal{C}[\![s]\!]([x \mapsto v_i])\big)(y) = v_o.$$

4.2 Baseline Search Algorithm

Algorithm 1 shows the basic architecture of our enumerative search algorithm. The algorithm initializes the workset W with s_0 (line 1). Then, it picks a state s with the

$$\frac{a \rightsquigarrow_a a'}{l := a \rightsquigarrow l := a'} \quad \frac{c_1 \rightsquigarrow c_1'}{c_1; c_2 \rightsquigarrow c_1'; c_2} \quad \frac{c_2 \rightsquigarrow c_2'}{c_1; c_2 \rightsquigarrow c_1; c_2'}$$

$$\frac{b \rightsquigarrow_b b'}{\mathit{if}\ b\ c_1\ c_2 \rightsquigarrow \mathit{if}\ b'\ c_1\ c_2} \quad \frac{c_1 \rightsquigarrow c_1'}{\mathit{if}\ b\ c_1\ c_2 \rightsquigarrow \mathit{if}\ b\ c_1'\ c_2} \quad \frac{c_2 \rightsquigarrow c_2'}{\mathit{if}\ b\ c_1\ c_2 \rightsquigarrow \mathit{if}\ b\ c_1\ c_2'}$$

$$\frac{b \rightsquigarrow_b b'}{\mathit{while}\ b\ c \rightsquigarrow \mathit{while}\ b'\ c} \quad \frac{c \rightsquigarrow c'}{\mathit{while}\ b\ c \rightsquigarrow \mathit{while}\ b\ c'}$$

$$\frac{b_1 \rightsquigarrow_b b_1'}{b_1 \wedge b_2 \rightsquigarrow_b b_1' \wedge b_2} \quad \frac{b_2 \rightsquigarrow_b b_2'}{b_1 \wedge b_2 \rightsquigarrow_b b_1 \wedge b_2'} \quad \frac{b_1 \rightsquigarrow_b b_1'}{b_1 \vee b_2 \rightsquigarrow_b b_1' \vee b_2} \quad \frac{b_2 \rightsquigarrow_b b_2'}{b_1 \vee b_2 \rightsquigarrow_b b_1 \vee b_2'} \quad \frac{b \rightsquigarrow_b b'}{\neg b \rightsquigarrow_b \neg b'}$$

$$\frac{}{\square \rightsquigarrow l := \Diamond} \quad \frac{}{\square \rightsquigarrow \mathit{skip}} \quad \frac{}{\square \rightsquigarrow \square; \square} \quad \frac{}{\square \rightsquigarrow \mathit{if}\ \triangle\ \square\ \square} \quad \frac{}{\square \rightsquigarrow \mathit{while}\ \triangle\ \square}$$

$$\frac{}{\triangle \rightsquigarrow_b \mathit{true}} \quad \frac{}{\triangle \rightsquigarrow_b \mathit{false}} \quad \frac{}{\triangle \rightsquigarrow_b l_1 \prec l_2} \quad \frac{}{\triangle \rightsquigarrow_b l \prec n}$$

$$\frac{}{\triangle \rightsquigarrow_b \triangle \wedge \triangle} \quad \frac{}{\triangle \rightsquigarrow_b \triangle \vee \triangle} \quad \frac{}{\triangle \rightsquigarrow_b \neg \triangle}$$

$$\frac{}{\Diamond \rightsquigarrow_a n} \quad \frac{}{\Diamond \rightsquigarrow_a l} \quad \frac{}{\Diamond \rightsquigarrow_a l_1 \oplus l_2} \quad \frac{}{\Diamond \rightsquigarrow_a l \oplus n}$$

Fig. 4. Transition Relation ($n \in \mathit{\Gamma}, l \in \mathbb{X}_i \cup \{x[y] \mid x \in \mathbb{X}_a \wedge y \in \mathbb{X}_i\}$)

smallest size and removes the state from the workset (line 3). We compute a size of a state using a heuristic cost model $\mathcal{M}_c : c \to \mathbb{Z}$ which is inductively defined as follows:

$$\begin{aligned}
\mathcal{M}_c(l := a) &= cost_1 + \mathcal{M}_a(a) \\
\mathcal{M}_c(\mathit{skip}) &= cost_2 \\
\mathcal{M}_c(c_1; c_2) &= cost_3 + \mathcal{M}_c(c_1) + \mathcal{M}_c(c_2) \\
\mathcal{M}_c(\mathit{while}\ b\ c) &= cost_4 + \mathcal{M}_b(b) + \mathcal{M}_c(c) \\
\mathcal{M}_c(\mathit{if}\ b\ c_1\ c_2) &= cost_5 + \mathcal{M}_b(b) + \mathcal{M}_c(c_1) + \mathcal{M}_c(c_2) \\
\mathcal{M}_c(\square) &= cost_6
\end{aligned}$$

where integer constants from $cost_1$ to $cost_6$ represent costs related to each command. After computing the sizes of all the states in the workset W, we pick the smallest state since the smaller ones are likely to generalize behavior better while avoiding overfitting for the given examples. Therefore, our current implementation prefers programs without holes to programs with holes (i.e., $cost_6 > cost_1, ..., cost_5$). Likewise, the cost models for arithmetic expressions ($\mathcal{M}_a : a \to \mathbb{Z}$) and boolean expressions ($\mathcal{M}_b : b \to \mathbb{Z}$) are also defined to prefer expressions without holes to expressions with holes.

If s is a solution state, the algorithm terminates and s is returned (line 5). For a non-terminal state, the algorithm attempts to prune the state by invoking the function prune (line 7). If pruning fails, the next states of s are added into the workset and the loop repeats. The details of our pruning technique is described in Section 4.3. At the moment, assume prune always fails.

The baseline algorithm implicitly performs well-known optimization techniques; it normalizes states in order to avoid exploring syntactically different but semantically the same ones. For instance, suppose we are exploring the state $(r := 0; r := x * 0; \square)$. We

Algorithm 1 Synthesis Algorithm

Input: A synthesis problem $((x, c_0, y), \mathcal{E}, \Gamma, \mathbb{X}_i, \mathbb{X}_a)$
Output: A complete program consistent with $\mathcal{E}$

```
1: W ← {s_0} where s_0 = c_0
2: repeat
3:     Pick the smallest state s from W
4:     if s is a terminal state then
5:         if solution(s) then return (x, s, y)
6:     else
7:         if ¬prune(s) then W ← W ∪ next(s)
8: until W = ∅
```

normalize it to $(r := 0; \square)$ and add it to the workset only when the resulting state has not been processed before. To do so, we first maintain previously explored states and never reconsider them. Secondly, we use four code optimization techniques: constant propagation, copy propagation, dead code elimination, and expression simplification [1]. For example, starting from $(r := 0; r := x * 0; \square)$, we simplify the expression $(x * 0)$ and obtain $(r := 0; r := 0; \square)$. Then, we apply dead code elimination to remove the first assignment $(r := 0; \square)$. Lastly, we also reorder variables in a fixed order. For example, when we assume alphabetical order, $(x := b + a)$ is rewritten as $(x := a + b)$. These normalization techniques significantly improve the speed of enumerative search.

In addition, the algorithm considers terminating programs only. Our language has unrestricted loops, so the basic algorithm may synthesize non-terminating programs. To exclude them from the search space, we use syntactic heuristics to detect potentially non-terminating loops. The heuristics are: 1) we only allow boolean expressions of the form $x < y$ (or $x > n$) in loop conditions, 2) the last statement of the loop body must increase (or decrease) the induction variable x, and 3) x and y are not defined elsewhere in the loop. If the states belong to one of the three cases, we prune the states. For example, we prune the state $(r := 0; \mathit{while}\ (x > 0)\{x := \Diamond; \square; x := x - 1\})$ since the induction variable x will be defined at the beginning of the loop body (i.e., the state violated the third rule).

4.3 Pruning with Static Analysis

Now we present the main contribution of this paper, pruning with static analysis. Static analysis allows to safely identify states that eventually fail to be a solution. We first define the notion of failure states.

Definition 1. *A state s is a failure state, denoted* $\mathsf{fail}(s)$*, iff every terminal state s' reachable from s is not a solution, i.e.,*

$$\mathsf{fail}(s) \iff ((s \rightsquigarrow^* s') \wedge s' \not\rightsquigarrow \implies \neg\mathsf{solution}(s')).$$

Our goal is to detect as many failure states as possible. We observed two typical cases of failure states that often show up during the baseline search algorithm.

```
example1(n){
  r := 0;
  while (n > 0){
    r := n + 1;
    n := ◊;
  };
  return r;
}
```

(a)

```
example2(n) {
  r := 0;
  while (n > 0){
        □;
    r := x * 10;
    n := n / 10;
  };
  return r;
}
```

(b)

Fig. 5. States that are pruned away

Example 2. Consider the program in Figure 5(a) and input-output example $(1, 1)$. When the program is executed with $n = 1$, no matter how the hole $(\Diamond)$ gets instantiated, the output value r is no less than 2 at the return statement. Therefore, the program cannot but fail to satisfy the example $(1, 1)$.

Example 3. Consider the program in Figure 5(b) and input-output example $(1, 1)$. Here, we do not know the exact values of x and r, but we know that $10 * x = 1$ must hold at the end of the program. However, there exists no such integer x, and we conclude the partial program is a failure state.

Static Analysis We designed a static analysis that aims to effectively identify these two types of failure states. To do so, our analysis combines numeric and symbolic analyses; the numeric analysis is designed to detect the cases of Example 2 and the symbolic analysis for the cases of Example 3. The abstract domain of the analysis is defined as follows:

$$\widehat{m} \in \widehat{\mathbb{M}} = \mathbb{X} \to \widehat{\mathbb{V}}, \quad \widehat{v} \in \widehat{\mathbb{V}} = \mathbb{I} \times \mathbb{S}$$

An abstract memory state $\widehat{m}$ maps variables to abstract values ($\widehat{\mathbb{V}}$). An abstract value is a pair of intervals ($\mathbb{I}$) and symbolic values ($\mathbb{S}$). The domain of intervals is standard [6]:

$$\mathbb{I} = (\{\bot\} \cup \{[l, u] \mid l, u \in \mathbb{Z} \cup \{-\infty, +\infty\} \wedge l \leq u\}, \sqsubseteq_{\mathbb{I}}).$$

For symbolic analysis, we define the following flat domain:

$$\mathbb{S} = (\mathsf{SE}^{\top}_{\bot}, \sqsubseteq_{\mathbb{S}}) \ \textit{where} \ \mathsf{SE} ::= \ n \mid \beta_x \ (x \in \mathbb{X}_i) \mid \mathsf{SE} \oplus \mathsf{SE}$$

A symbolic expression $se \in \mathsf{SE}$ is an integer (n), a symbol (β_x), or a binary operation with symbolic expressions. We introduce symbols one for each integer-type variable in the program. The symbolic domain is flat and has the partial order: $s_1 \sqsubseteq_{\mathbb{S}} s_2 \iff (s_1 = \bot) \vee (s_1 = s_2) \vee (s_2 = \top)$. We define the abstraction function $\alpha : \mathbb{V} \to \widehat{\mathbb{V}}$ that transforms concrete values to abstract values:

$$\begin{aligned} \alpha(n) &= ([n, n], n) \\ \alpha(n_1 \ldots n_k) &= ([\min\{n_1, \ldots, n_k\}, \max\{n_1, \ldots, n_k\}], \top). \end{aligned}$$

$$
\begin{array}{rl}
\widehat{\mathcal{A}}[\![n]\!](\widehat{m}) &= ([n,n],n) \\
\widehat{\mathcal{A}}[\![l]\!](\widehat{m}) &= \widehat{m}(x)\ (l = x \text{ or } x[y]) \\
\widehat{\mathcal{A}}[\![l_1 \oplus l_2]\!](\widehat{m}) &= \widehat{\mathcal{A}}[\![l_1]\!](\widehat{m}) \mathbin{\widehat{\oplus}} \widehat{\mathcal{A}}[\![l_2]\!](\widehat{m}) \\
\widehat{\mathcal{A}}[\![l \oplus n]\!](\widehat{m}) &= \widehat{\mathcal{A}}[\![l_1]\!](\widehat{m}) \mathbin{\widehat{\oplus}} n \\
\widehat{\mathcal{A}}[\![\Diamond]\!](\widehat{m}) &= ([-\infty,+\infty],\top) \\
\widehat{\mathcal{B}}[\![true]\!](\widehat{m}) &= \widehat{true} \\
\widehat{\mathcal{B}}[\![false]\!](\widehat{m}) &= \widehat{false} \\
\widehat{\mathcal{B}}[\![l_1 \prec l_2]\!](\widehat{m}) &= \widehat{\mathcal{A}}[\![l_1]\!](\widehat{m}) \mathbin{\widehat{\prec}} \widehat{\mathcal{A}}[\![l_2]\!](\widehat{m}) \\
\widehat{\mathcal{B}}[\![l \prec n]\!](\widehat{m}) &= \widehat{\mathcal{A}}[\![l]\!](\widehat{m}) \mathbin{\widehat{\prec}} n \\
\widehat{\mathcal{B}}[\![b_1 \wedge b_2]\!](\widehat{m}) &= \widehat{\mathcal{B}}[\![b_1]\!](\widehat{m}) \mathbin{\widehat{\wedge}} \widehat{\mathcal{B}}[\![b_2]\!](\widehat{m}) \\
\widehat{\mathcal{B}}[\![b_1 \vee b_2]\!](\widehat{m}) &= \widehat{\mathcal{B}}[\![b_1]\!](\widehat{m}) \mathbin{\widehat{\vee}} \widehat{\mathcal{B}}[\![b_2]\!](\widehat{m}) \\
\widehat{\mathcal{B}}[\![\neg b]\!](\widehat{m}) &= \widehat{\neg}\widehat{\mathcal{B}}[\![b]\!](\widehat{m}) \\
\widehat{\mathcal{B}}[\![\triangle]\!](\widehat{m}) &= \top
\end{array}
$$

$$
\begin{array}{rl}
\widehat{\mathcal{C}}[\![x := \Diamond]\!](\widehat{m}) &= \widehat{m}[x \mapsto ([-\infty,+\infty],\beta_x)] \\
\widehat{\mathcal{C}}[\![x[y] := \Diamond]\!](\widehat{m}) &= \widehat{m}[x \mapsto ([-\infty,+\infty],\top)] \\
\widehat{\mathcal{C}}[\![x := a]\!](\widehat{m}) &= \widehat{m}[x \mapsto \widehat{\mathcal{A}}[\![a]\!](\widehat{m})] \\
\widehat{\mathcal{C}}[\![x[y] := a]\!](\widehat{m}) &= \widehat{m}[x \mapsto \widehat{\mathcal{A}}[\![a]\!](\widehat{m}) \sqcup \widehat{m}(x)] \\
\widehat{\mathcal{C}}[\![skip]\!](\widehat{m}) &= \widehat{m} \\
\widehat{\mathcal{C}}[\![c_1; c_2]\!](\widehat{m}) &= (\widehat{\mathcal{C}}[\![c_2]\!] \circ \widehat{\mathcal{C}}[\![c_1]\!])(\widehat{m}) \\
\widehat{\mathcal{C}}[\![if\ b\ c_1\ c_2]\!](\widehat{m}) &= \widehat{\mathsf{cond}}(\widehat{\mathcal{B}}[\![b]\!], \widehat{\mathcal{C}}[\![c_1]\!], \widehat{\mathcal{C}}[\![c_2]\!])(\widehat{m}) \\
\widehat{\mathcal{C}}[\![while\ b\ c]\!](\widehat{m}) &= (\widehat{\mathsf{fix}}\ \widehat{F})(\widehat{m}) \\
\text{where } \widehat{F}(g) &= \widehat{\mathsf{cond}}(\widehat{\mathcal{B}}[\![b]\!], g \circ \widehat{\mathcal{C}}[\![c]\!], \lambda x.x) \\
\widehat{\mathcal{C}}[\![\Box]\!](\widehat{m})(x) &= \begin{cases} ([-\infty,+\infty],\beta_x) & x \in \mathbb{X}_i \\ ([-\infty,+\infty],\top) & x \in \mathbb{X}_a \end{cases} \\
\widehat{\mathsf{cond}}(p,f,g)(\widehat{m}) &= \begin{cases} \bot & \text{if } p(\widehat{m}) = \bot \\ f(\widehat{m}') & \text{if } p(\widehat{m}) = \widehat{true} \\ g(\widehat{m}') & \text{if } p(\widehat{m}) = \widehat{false} \\ f(\widehat{m}') \sqcup g(\widehat{m}') & \text{if } p(\widehat{m}) = \top \end{cases} \\
\text{where } \widehat{m}' &= \bigsqcup\{\widehat{m}'' \sqsubseteq_{\widehat{\mathbb{M}}} \widehat{m} \mid \widehat{m}'' \models p\}
\end{array}
$$

Fig. 6. Abstract semantics

The abstract semantics is defined in Figure 6 by the functions:

$$\widehat{\mathcal{A}}[\![a]\!] : \widehat{\mathbb{M}} \to \widehat{\mathbb{V}}, \quad \widehat{\mathcal{B}}[\![b]\!] : \widehat{\mathbb{M}} \to \widehat{\mathbb{B}}, \quad \widehat{\mathcal{C}}[\![c]\!] : \widehat{\mathbb{M}} \to \widehat{\mathbb{M}}$$

where $\widehat{\mathbb{B}} = \{\widehat{true}, \widehat{false}\}^{\top}_{\bot}$ is the abstract boolean lattice.

Intuitively, the abstract semantics over-approximates the concrete semantics of *all* terminal states that are reachable from the current state. This is done by defining the sound semantics for holes: $\widehat{\mathcal{A}}[\![\Diamond]\!](\widehat{m})$, $\widehat{\mathcal{B}}[\![\triangle]\!](\widehat{m})$, and $\widehat{\mathcal{C}}[\![\Box]\!](\widehat{m})$. An exception is that integer variables get assigned symbols, rather than $\top$, in order to generate symbolic constraints on integer variables.

In our analysis, array elements are abstracted into a single element. Hence, the definitions of $\widehat{\mathcal{A}}[\![x[y]]\!]$ and $\widehat{\mathcal{C}}[\![x[y] := a]\!]$ do not involve y. Because an abstract array cell may represent multiple concrete cells, arrays are weakly updated by joining ($\sqcup$) old and new values. For example, given a memory state $\widehat{m} = [x \mapsto ([5,5],\top), ...]$, $\widehat{\mathcal{C}}[\![x[y] := 1]\!](\widehat{m})$ evaluates to $[x \mapsto ([1,5],\top), ...]$.

For while-loops, the analysis performs a sound fixed point computation. If the computation does not reach a fixed point after a fixed number of iterations, we apply widening for infinite interval domain, in order to guarantee the termination of the analysis. We use the standard widening operator in [6]. The function $\widehat{\mathsf{fix}}$ and $\widehat{\mathsf{cond}}$ in Figure 6 denote a post-fixed point operator and a sound abstraction of cond, respectively.

Pruning Next we describe how we do pruning with the static analysis. Suppose we are given examples $\mathcal{E} \subseteq \mathbb{V} \times \mathbb{V}$ and a state s with input (x) and output (y) variables. For each example $(v_i, v_o) \in \mathcal{E}$, we first run the static analysis with the input $\alpha(v_i)$ and obtain the analysis result (itv_s, se_s) :

$$(itv_s, se_s) = \big(\widehat{\mathcal{C}}[\![s]\!]([x \mapsto \alpha(v_i)])\big)(y).$$

We only consider the case when $itv_s = [l_s, u_s]$ (when $itv_s = \bot$, the program is semantically ill-formed and therefore we just prune out the state). Then, we obtain the interval abstraction $[l_o, u_o]$ of the output v_o, i.e., $([l_o, u_o], -) = \alpha(v_o)$, and generate the constraints $C^s_{(v_i,v_o)}$:

$$C^s_{(v_i,v_o)} = (l_s \leq l_o \wedge u_o \leq u_s) \wedge (se_s \in \mathsf{SE} \implies l_o \leq se_s \leq u_o).$$

The first (resp., second) conjunct means that the interval (resp., symbolic) analysis result must over-approximate the output example. We prune out a state s iff $C^s_{(v_i,v_o)}$ is unsatisfiable for some example $(v_i, v_o) \in \mathcal{E}$:

Definition 2. *The predicate* prune *is defined as follows:*

$$\mathsf{prune}(s) \iff C^s_{(v_i,v_o)} \textit{ is unsatisfiable for some } (v_i, v_o) \in \mathcal{E}.$$

The unsatisfiability can be easily checked, for instance, with an off-the-shelf SMT solver. Example 4 and 5 show how the above pruning works.

Example 4. Consider the Example 2 again, where $(1, 1) \in \mathcal{E}$. Let s be the state in Figure 5(a). If we run the analysis with the input $\alpha(1)$, we get

$$([2, +\infty], \top) = \big(\widehat{\mathcal{C}}[\![s]\!]([n \mapsto \alpha(1)])\big)(r).$$

To see why, each time the loop-body is executed, the interval value of n becomes $[-\infty, +\infty]$, and the interval value of r is stabilized at $[2, +\infty]$ after the third execution of the loop-body. Also, the resulting symbolic value of r is $\top$, because if we join the first loop-body execution result 2 and the second loop-body execution result $\beta_n + 1$, the resulting value becomes $\top$ and stabilized as $\top$ thereafter. As a result, we get the constraint

$$C^s_{(v_i,v_o)} = (2 \leq 1 \wedge 1 \leq +\infty)$$

since $(itv_s, se_s) = ([2, +\infty], \top)$, $[l_o, u_o] = [1, 1]$, and $\top \notin \mathsf{SE}$. The constraint is unsatisfiable since $2 \leq 1$ is never true. Hence, we prune out the state s.

Example 5. Consider the Example 3 again, where $(1, 1) \in \mathcal{E}$. Let s be the state in Figure 5(b). If we run the analysis with the input $\alpha(1)$, we get

$$([-\infty, +\infty], \beta_x * 10) = \big(\widehat{\mathcal{C}}[\![s]\!]([n \mapsto \alpha(1)])\big)(r)$$

since, by the semantic computations of the command hole ($\square$), interval values of the variables become $[-\infty, +\infty]$ and symbolic value of r is fixed at $\beta_x * 10$. As a result, we get the constraint

$$C^s_{(v_i,v_o)} = (-\infty \leq 1 \wedge 1 \leq +\infty) \wedge (\beta_x * 10 \in \mathsf{SE} \implies 1 \leq \beta_x * 10 \leq 1)$$

where $(itv_s, se_s) = ([-\infty, +\infty], \beta_x * 10)$ and $[l_o, u_o] = [1, 1]$. This constraint is unsatisfiable since $\beta_x * 10$ is in SE (i.e., $\beta_x * 10 \in \mathsf{SE}$), but no β_x exists such that $1 \leq \beta_x * 10 \leq 1$ holds. Therefore, we prune out the state s.

The following theorem states that our pruning is safe:

Theorem 1 (Safety). $\forall s \in S.\ \mathsf{prune}(s) \implies \mathsf{fail}(s)$.

That is, we prune out a state only when it is a failure state, which formally guarantees that the search algorithm with our pruning finds a solution if and only if the baseline algorithm (Section 4.2) does so.

5 Evaluation

To evaluate our synthesis algorithm, we gathered 30 introductory level problems from several online forums [2] (Table 1). All of the benchmark problems we used are publicly available with our tool SIMPL[3]. The problems consist of various tasks of manipulating integers and arrays of integers. Some problems are non-trivial for novice programmers to solve; the problems require the novices to come up with various control structures such as nested loops and combinations of loops and conditional statements.

For all but three problems (#23, #24, #29), we used partial programs similar to those in Figure 1(a)–(c), which consist of initialization statements followed by a single loop with empty condition and body: e.g.,

```
problem (n) { r := 0; while (?) { ? }; return r; }
```

That is, in most cases, the synthesis goal is to complete the loop condition and body. For the other problems (#23, #24, #29), we used partial programs similar to the one in Figure 1(d), where the job is to complete the condition and body of conditional statements. For instance, for problem #23, we used the following template on the left-hand side. The synthesized program is given on the right-hand side.

```
problem23 (arr, len) {
  i := 0;
  m := arr[i];
  while (i < len) {
    if (?) { ? };
    i=i+1;
  };
  return m;
}
```

```
problem23 (arr, len) {
  i := 0;
  m := arr[i];
  while (i < len) {
    if (arr[i]>m) {m:=arr[i]};
    i=i+1;
  };
  return m;
}
```

For each benchmark, we report the number of integer-type variables (the column 'IVars'), array-type variables (the column 'AVars'), integer constants (the column 'Ints'), and input-output examples (the column 'Exs') provided, respectively. To show practicality of SIMPL, we gave over-estimated resources ('IVars', 'AVars', 'Ints') for some benchmarks, provided small number (2–4) of input-output examples, and configured the examples to be easy even for the beginners to come up with, as shown in Section 2. All of the experiments were conducted on MacBook Pro with Intel Core i7 and 16GB of memory.

Table 1 shows the performance of our algorithm. The column 'Enum' shows the running time of enumerative search without state normalization. In that case, the average runtime was longer than 616 seconds, and three of the benchmarks timed out (> 1 hour). The column 'Base' reports the performance of our baseline algorithm with normalization. It shows that normalizing states succeeds to solve all benchmark problems and improves the speed by more than 3.7 times on average, although it degrades the speed for some cases due to normalization runtime overhead.

[2] E.g., http://www.codeforwin.in

[3] http://prl.korea.ac.kr/simpl

Table 1. Performance of SIMPL. $\perp$ denotes timeout ($>$ 1 hour). Assume $\perp$ as 3,600 seconds for the average of 'Enum'.

	No	Description	Vars		Ints	Exs	Time (sec)		
			IVars	AVars			Enum	Base	Ours
Integer	1	Given n, return $n!$.	2	0	2	4	0.0	0.0	0.0
	2	Given n, return $n!!$.	3	0	3	4	0.0	0.0	0.0
	3	Given n, return $\sum_{i=1}^{n} i$.	3	0	2	4	0.1	0.0	0.0
	4	Given n, return $\sum_{i=1}^{n} i^2$.	4	0	2	3	122.4	18.1	0.3
	5	Given n, return $\prod_{i=1}^{n} i^2$.	4	0	2	3	102.9	13.6	0.2
	6	Given a and n, return a^n.	4	0	2	4	0.7	0.1	0.1
	7	Given n and m, return $\sum_{i=n}^{m} i$.	3	0	2	3	0.2	0.0	0.0
	8	Given n and m, return $\prod_{i=n}^{m} i$.	3	0	2	3	0.2	0.0	0.1
	9	Count the digits of an integer.	3	0	3	3	0.0	0.0	0.0
	10	Sum the digits of an integer.	3	0	3	4	5.2	2.2	1.3
	11	Find the product of each digit of an integer.	3	0	3	3	0.7	2.3	0.3
	12	Count the number of binary digit of an integer.	2	0	3	3	0.0	0.0	0.0
	13	Find the nth Fibonacci number.	3	0	3	4	98.7	13.9	2.6
	14	Given n, return $\sum_{i=1}^{n}(\sum_{m=1}^{i} m))$.	3	0	2	4	$\perp$	324.9	37.6
	15	Given n, return $\prod_{i=1}^{n}(\prod_{m=1}^{i} m))$.	3	0	2	4	$\perp$	316.6	86.9
	16	Reverse a given integer.	3	0	3	3	$\perp$	367.3	2.5
Array	17	Find the sum of all elements of an array.	3	1	2	2	8.1	3.6	0.9
	18	Find the product of all elements of an array.	3	1	2	2	7.6	3.9	0.9
	19	Sum two arrays of same length into one array.	3	2	2	2	44.6	29.9	0.2
	20	Multiply two arrays of same length into one array.	3	2	2	2	47.4	26.4	0.3
	21	Cube each element of an array.	3	1	1	2	1283.3	716.1	13.0
	22	Manipulate each element of an array into fourth power.	3	1	1	2	1265.8	715.5	13.0
	23	Find a maximum element.	3	1	2	2	0.9	0.7	0.4
	24	Find a minimum element.	3	1	2	2	0.8	0.3	0.1
	25	Add 1 to each element.	2	1	1	3	0.3	0.0	0.0
	26	Find the sum of square of each element.	3	1	2	2	2700.0	186.2	11.5
	27	Find the product of square of each element.	3	1	1	2	1709.8	1040.3	12.6
	28	Sum the products of matching elements of two arrays.	3	2	1	3	20.5	38.7	1.5
	29	Sum the absolute values of each element.	2	1	1	2	45.0	50.5	12.1
	30	Count the number of each element.	3	1	3	2	238.9	1094.1	0.2
	Average						$>$ 616.8	165.5	6.6

On top of 'Base', we applied our static-analysis-guided pruning technique (the column 'Ours'). The results show that our pruning technique is remarkably effective. It reduces the average time to 6.6 seconds, improving the speed of 'Base' by 25 times and the speed of 'Enum' by more than 93 times. We manually checked that all of the synthesized solutions are correct. We also found that all the solutions are quite intuitive and instructive as demonstrated in Section 2.

6 Related Work

In this section, we broadly survey recent approaches in program synthesis. Most importantly, our work is different from existing program synthesis techniques in that we combine enumerative program synthesis with semantic-based static analysis.

Program Synthesis Techniques We compare our algorithm with three most prevalent synthesis approaches: enumerative synthesis, version space algebra (VSA), and solver-aided method.

Enumerative synthesis has been widely used to synthesize recursive functional programs [2, 8, 9, 24, 26], API-completion [22, 13, 25, 12, 7], tree-structured data transformation [33], and regular expressions [21]. ESCHER [2] uses a heuristic goal-directed search to synthesize functional programs. Unlike ours, their algorithm finds smaller programs that partially satisfy given examples, and combines partial solutions with if-then-else statements. Although effective for recursive programs, it may cause overfitting in our case. In [8, 24, 9, 26, 22, 13, 25, 12, 7], type systems are used to exclude ill-typed programs from search space (i.e., type-directed synthesis). However, type-based pruning is not applicable to ours because enumerated terms are all well-typed. This is also the case for regular expression synthesis [21]. In λ^2 [8], deduction is also used in order to reject partial programs that are logically inconsistent with input-output examples. Feser et al. [8] designed a set of deduction rules for higher-order components such as `map` and `fold`. However, the deduction approach is not applicable to ours; it is completely nontrivial to design useful deduction rules for programming constructs such as while-loop and conditional. ALPHAREGEX [21] performs over- and under approximations on partial regular expressions to prune them when further search cannot produce any solutions. But those pruning techniques are specialized for regular expression synthesis. HADES [33] uses SMT solvers and decision tree learning to perform path transformations, but again it is not appropriate for imperative program synthesis. To sum up, we cannot use existing pruning techniques and in this paper we show that using static value analysis is a promising alternative for synthesizing imperative programs.

Many program synthesis approaches using version space algebra (VSA) have been proposed for string manipulation [10, 17, 23, 27, 32], number transformation [28], extracting data [4, 20], and data filtering [31]. VSA is a kind of divide-and-conquer search strategy where a solution program is constructed by combining the solutions to sub-problems (e.g., some portions of the examples) in a top-down way. In contrast, we do not look for a sub-solution for each of the sub-problems, but instead in a bottom-up way, we find a total solution that satisfies all the examples at once.

Solver-aided methods have also been used many times to synthesize recursive functions [18], dynamic programming implementations [15], loop-free programs for bit-vector applications [11], and low-level missing details for programmers [29]. They use counter-example guided inductive synthesis (CEGIS) which iteratively refines the target concept from the counter-examples provided by SMT/SAT solver until the solution is verified by the solver. We use solvers to check the satisfiability of the symbolic constraints generated by the static analysis, not to refine the search space based on the counter-examples.

Additionally, DeepCoder [3] uses deep learning to guide search in program synthesis. In DeepCoder, a probabilistic distribution is learned to predict the presence or absence of each function in domain-specific languages. For example, given an input-output list $[-17, -3, 4, 11] \mapsto [-12, -68]$, DeepCoder learns that a list-filtering function is likely to be involved in the resulting program, since the number of elements in the input list is reduced. This idea of learning to rank programs is orthogonal to our approach. In [16], model checking was applied to guide genetic algorithm. Katz and Peled used model checking in computing fitness function, which computes fitness score of each candidate. Essentially, however, the genetic algorithm does not guarantee finding a solution unlike enumerative synthesis approach.

Imperative Program Synthesis There has been little work on imperative program synthesis. In 2003, Lau et al. [19] proposed an approach to learning programs in a subset of imperative Python using version space algebra. However, the system requires a value trace for each program point as input (i.e., programming by demonstration), which is unrealistic to be used. In 2005, Colón [5] presented a schema-guided synthesis of imperative programs from pre- and post condition that compute polynomial functions where the programs can only be generated from a collection of available schemas, which has an inherent disadvantage of incompleteness. In 2006, Ireland et al. [14] proposed an approach to constructing imperative programs from logical assertions by leveraging theorem proving technique. We believe that it is a good direction to use theorem proving techniques with lightweight logical specifications as inputs for future work in order to synthesize more complicated programs. In 2010, Srivastava et al. [30], the most recent work on imperative program synthesis to the best of our knowledge, presented a view that treats a synthesis problem as verification problem. The researchers showed that they can synthesize complex tasks such as sorting and dynamic programming. Although inspiring, their system's ability is limited to underlying program verifiers that solve given synthesis conditions, thus more efficient verifiers need to be developed first in order to deal with more complicated synthesis tasks.

7 Conclusion

In this paper, we have shown that combining enumerative synthesis with static analysis is a promising way of synthesizing imperative programs. The enumerative search allows us to find the smallest possible, therefore general, program while the semantic-based static analysis dramatically accelerates the search in a safe way. We demonstrated the effectiveness on 30 introductory programming problems gathered from online forums.

Acknowledgements We appreciate the anonymous reviewers for their helpful comments. This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP) (No.2017-0-00184, Self-Learning Cyber Immune Technology Development) This research was also supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT & Future Planning(NRF-2016R1C1B2014062).

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
2. Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *CAV*, 2013.
3. Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *ICLR*, 2017.
4. Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. Flashrelate: Extracting relational data from semi-structured spreadsheets using examples. In *PLDI*, 2015.
5. Michael A. Colón. Schema-guided synthesis of imperative programs by constraint solving. In *LOPSTR*, 2005.
6. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
7. Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. Component-based synthesis for complex apis. In *POPL*, 2017.
8. John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *PLDI*, 2015.
9. Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: A type-theoretic interpretation. In *POPL*, 2016.
10. Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011.
11. Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.
12. Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In *PLDI*, 2013.
13. Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac. Interactive synthesis of code snippets. In *CAV*, 2011.
14. Andrew Ireland and Jamie Stark. Combining proof plans with partial order planning for imperative program synthesis. *Automated Software Eng.*, 2006.
15. Shachar Itzhaky, Rohit Singh, Armando Solar-Lezama, Kuat Yessenov, Yongquan Lu, Charles Leiserson, and Rezaul Chowdhury. Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations. In *OOPSLA*, 2016.
16. Gal Katz and Doron Peled. Synthesizing, correcting and improving code, using model checking-based genetic programming. In *HVC*, 2013.
17. Dileep Kini and Sumit Gulwani. Flashnormalize: Programming by examples for text normalization. In *IJCAI*, 2015.
18. Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. Synthesis modulo recursive functions. In *OOPSLA*, 2013.
19. Tessa Lau, Pedro Domingos, and Daniel S. Weld. Learning programs from traces using version space algebra. In *K-CAP*, 2003.

20. Vu Le and Sumit Gulwani. Flashextract: A framework for data extraction by examples. In *PLDI*, 2014.
21. Mina Lee, Sunbeom So, and Hakjoo Oh. Synthesizing regular expressions from examples for introductory automata assignments. In *GPCE*, 2016.
22. David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: Helping to navigate the api jungle. In *PLDI*, 2005.
23. Mehdi Manshadi, Daniel Gildea, and James Allen. Integrating programming by example and natural language programming. In *AAAI*, 2013.
24. Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *PLDI*, 2015.
25. Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed completion of partial expressions. In *PLDI*, 2012.
26. Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *PLDI*, 2016.
27. Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. Compositional program synthesis from natural language and examples. In *IJCAI*, 2015.
28. Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *CAV*, 2012.
29. Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, Berkeley, CA, USA, 2008.
30. Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *POPL*, 2010.
31. Xinyu Wang, Sumit Gulwani, and Rishabh Singh. Fidex: Filtering spreadsheet data using examples. In *OOPSLA*, 2016.
32. Bo Wu and Craig A. Knoblock. An iterative approach to synthesize data transformation programs. In *IJCAI*, 2015.
33. Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. Synthesizing transformations on hierarchically structured data. In *PLDI*, 2016.