Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)

Synthesizing Pattern Programs from Examples

Sunbeom So
Korea University
sunbeom_so@korea.ac.kr

Hakjoo Oh
Korea University
hakjoo_oh@korea.ac.kr

Abstract

We describe a programming-by-example system that automatically generates pattern programs from examples. Writing pattern programs, which produce various patterns of characters, is one of the most popular programming exercises for entry-level students. However, students often find it difficult to write correct solutions by themselves. In this paper, we present a method for synthesizing pattern programs from examples, allowing students to improve their programming skills efficiently. To that end, we first design a domain-specific language that supports a large class of pattern programs that students struggle with. Next, we develop a synthesis algorithm that efficiently finds a desired program by combining enumerative search, constraint solving, and program analysis. We implemented the algorithm in a tool and evaluated it on 40 exercises gathered from online forums. The experimental results and user study show that our tool can synthesize instructive solutions from 1–3 example patterns in 1.2 seconds on average.

1 Introduction

Writing pattern programs is one of the most popular programming exercises for novice programmers, where the task is to write a program that produces patterns of characters or stars that are arranged according to some logical rules. Examples include geometric patterns such as triangles, squares, and parallelograms.

Learning to write pattern programs in introductory programming courses is important because of two reasons. First, pattern programs provide excellent opportunities for enhancing key programming skills that involve nontrivial nested loops and conditional statements. Secondly, students can acquire logical thinking by trying to discover hidden rules in the pattern examples.

In this paper, we describe a programming-by-example system that automatically learns pattern programs from examples. Beginners often find it difficult to write correct pattern programs by themselves. Unfortunately, manually providing guidance in online/offline classes simply does not scale for the increasingly large number of students. Our goal is to build an automated tool that can help students to improve their programming skills without human teachers.

To that end, we first studied various online forums and designed a domain-specific language that supports a large class of pattern programs that students struggle with. Next, we developed a novel synthesis algorithm that effectively combines enumerative search, constraint solving, and program analysis. The crux of our algorithm is the automatic provision of program components that are required to write correct programs. We demonstrate this capability in Section 2.

We implemented our approach in a tool, PAT, and evaluated it on 40 benchmark problems gathered from online forums. Some of the benchmarks are challenging even for skilled programmers, requiring them to discover complex rules hidden in the patterns. The experimental results show that our approach is remarkably efficient; PAT can synthesize the desired programs only from 1–3 example patterns in 1.2 seconds on average. Further, our user study with 23 undergraduate students shows that PAT can aid them by providing instructive code; 91% responded that programs generated by PAT have qualities that are higher than or similar to human-written code, in terms of simplicity and readability.

We summarize our contributions below:

- We present a new programming-by-example system that automatically synthesizes pattern programs from examples. We make our tool and data publicly available.[1]
- We present a novel synthesis algorithm that combines three techniques: enumerative search, constraint solving, and program analysis. This combination enables PAT to efficiently synthesize pattern programs.
- We provide detailed evaluations of the tool with 40 pattern tasks from online forums and 23 students.

2 Illustrative Examples

In this section, we illustrate our tool, PAT, with two programming problems gathered from online forums.

To use our tool, it suffices to provide example patterns only. Without any other hints from a user, PAT is able to generate a program that reproduces the patterns while generalizing the behavior beyond the given examples.

[1] http://prl.korea.ac.kr/pat

Problem 1 (Isosceles Triangle) The first problem is to write a program that displays isosceles triangles (i.e., triangle with two equal sides). One possible example pattern is as follows:

Given this pattern, PAT takes 0.04 seconds to synthesize a program below that draws the pattern:

```
for i in N do:
  for j in N − i do: print ␣
  for j in 2 ∗ i − 1 do: print ★
  print ↩
```

The outermost loop iterates through the rows of the triangle, where N represents the number of the rows to be displayed. In each iteration of the rows, two inner loops are used to print $N - i$ blanks (␣) followed by $2 * i - 1$ star symbols (★). Also, the program puts a newline (↩) upon completion of each row. In this case, PAT can generate the program from a single example pattern.

Problem 2 (Hollow square with diagonals) Consider the problem of generating hollow squares with diagonals inside:

This problem has remained unanswered in an online forum[2] for seven months.

With PAT, however, we can solve this problem in 5.2s. Given the two examples above, PAT generates the following:

```
for i in N do:
  for j in N do:
    if ( i = 1 || i = N || j = 1 || j = i ||
         j = N − i + 1 || j = N): print ★
    else: print ␣
  print ↩
```

The program prints out ★ when one of the following conditions holds: the first row ($i = 1$), the last row ($i = N$), the first column ($j = 1$), the last column ($j = N$), the lower-right-oriented diagonal ($j = i$), and the upper-right-oriented diagonal ($j = N - i + 1$). PAT accomplishes this by introducing a conditional statement within the nested loops.

We finish this section by summarizing key features of PAT demonstrated through the examples:

- PAT is able to quickly generate instructive solutions for a broad range of pattern programming tasks, which allows students to benefit without asking questions and waiting responses in online forums for months.
- PAT does not require any hints from a user. PAT automatically infers program components such as integers necessary for writing the program. By contrast, existing program synthesizers typically require the user to manually provide such components, which is unrealistic for end-users. We provide detailed discussion in Section 6.
- PAT works with only few example patterns, minimizing the burden of writing examples on users. In our experiments, it was enough to provide 1–3 examples to obtain desired programs.

[2]http://codeforwin.org/2015/07/c-program-to-print-diamond-star-pattern.html, accessed 31-Jan-2018

3 Domain-Specific Language (DSL)

Figure 1 presents our domain-specific language (DSL) for describing patterns. In program synthesis, choosing a right DSL is a key to success [Gulwani *et al.*, 2017], as the DSL plays an important role in balancing expressiveness for a target domain and efficiency of a synthesis algorithm. General-purpose languages such as C are excessive for our purpose, only to impair the efficiency of the synthesis procedure. Thereby, we have carefully designed our DSL based on pattern programming tasks that novices struggle with by studying online forums.

Syntax We observed that a large number of pattern programs share a common structure and therefore made the following design decisions:

- A program (r) is a single loop (for i in N do c; ↩) that iterates through rows of the pattern, where each row is followed by a newline symbol (↩). Variables N and i are fixed over the program, which denote the number of rows and the iterator, respectively.
- The body (c) of the row-loop consists of a sequence of column loops. A column loop (for j in a do p) iterates through columns for a times. j is a fixed variable that is used in all column-loops.
- The body (p) of a column-loop consists of a symbol ($l \in \{★, ␣\}$) or a conditional expression (*if* b l_1 l_2).
- An arithmetic expression (a) is a linear expression ($x * N + y * i + z$) that only involves variables N and i, where x, y, and z are integer constants. At first glance, this seems overly restrictive, but most ★-patterns can be captured with this expression.
- A boolean expression (b) includes equalities for the beginning of the row ($i = 1$), the end of the row ($i = N$) and the column index ($j = a$), or a disjunction ($b_1 \mid\mid b_2$).

A program may have holes ($\diamondsuit, \triangle, \bigcirc, \square$) during the synthesis process, but the holes never appear in a final program.

Expressiveness Informally, our DSL is able to express patterns with following characteristics:

1. Patterns consisting of straight lines that are inclined at various angles (e.g., X-pattern in Problem 2).
2. Patterns consisting of subpatterns that are horizontally aligned (e.g., a subpattern ◸ is hidden on the left of the main pattern in Problem 1, drawn using blanks).

The first- and second features can be captured because, in our DSL, arithmetic expressions are linear and programs can have multiple inner column-loops, respectively.

$$
\begin{aligned}
r &::= \mathsf{for}\ i\ \mathsf{in}\ N\ \mathsf{do}\ (c; \hookleftarrow) \\
c &::= \mathsf{for}\ j\ \mathsf{in}\ a\ \mathsf{do}\ p \mid c_1; c_2 \mid \square \\
p &::= l\ (l \in \{\bigstar, \_\}) \mid \mathit{if}\ b\ l_1\ l_2 \mid \bigcirc \\
a &::= x * N + y * i + z\ (x, y, z \in \mathbb{Z}) \mid \diamondsuit \\
b &::= i = 1 \mid i = N \mid j = a \mid b_1 \mid\mid b_2 \mid \triangle
\end{aligned}
$$

Figure 1: Syntax of the DSL

$$
\begin{aligned}
\mathcal{A}[\![x * N + y * i + z]\!](m) &= x * m(N) + y * m(i) + z \\
\mathcal{B}[\![i = 1]\!](m) &= \big(m(i) = 1\big) \\
\mathcal{B}[\![i = N]\!](m) &= \big(m(i) = m(N)\big) \\
\mathcal{B}[\![j = a]\!](m) &= \big(m(j) = \mathcal{A}[\![a]\!](m)\big) \\
\mathcal{B}[\![b_1 \mid\mid b_2]\!](m) &= \mathcal{B}[\![b_1]\!](m) \vee \mathcal{B}[\![b_2]\!](m) \\
\mathcal{P}[\![l]\!](m) &= l\ (l \in \{\bigstar, \_\}) \\
\mathcal{P}[\![\mathit{if}\ b\ l_1\ l_2]\!](m) &= \begin{cases} l_1 & \text{if } \mathcal{B}[\![b]\!](m) = \mathit{true} \\ l_2 & \text{if } \mathcal{B}[\![b]\!](m) = \mathit{false} \end{cases} \\
\mathcal{C}[\![\mathsf{for}\ j\ \mathsf{in}\ a\ \mathsf{do}\ p]\!](m) &= \overset{\mathcal{A}[\![a]\!](m)}{\underset{k=1}{\bullet}} \mathcal{P}[\![p]\!](m[j \mapsto k]) \\
\mathcal{C}[\![c_1; c_2]\!](m) &= \big(\mathcal{C}[\![c_1]\!](m)\big)\big(\mathcal{C}[\![c_2]\!](m)\big)
\end{aligned}
$$

Figure 2: Semantics of the DSL

However, PAT may not be able to write patterns in which subpatterns are vertically aligned, since those patterns usually require multiple outer loops while our program has only one outer loop. Nevertheless, PAT can be useful in those cases, too. For example, imagine a pattern where an upper half is a triangle and a lower half is a square. Even if a student does not know how to write a corresponding solution at once, it is relatively easy to observe that different pattern rules may be needed to implement each of two subpatterns (triangle and square) and thereby two outer loops may be needed to write a total solution. Then, based on this observation, the student can get a final solution by separately applying PAT to each subpattern and combining the results.

Semantics A program in our DSL evaluates to a string over $\Sigma = \{\bigstar, \_, \hookleftarrow\}$. The semantics is formally defined in Figure 2. Let $\mathbb{X}$ be a set of program variables, i.e., $\mathbb{X} = \{i, j, N\}$. A memory state m is a partial function from variables to integers ($\mathbb{Z}$), i.e., $m \in \mathbb{M} = \mathbb{X} \to \mathbb{Z}$. Then, the semantics of the DSL is defined by the functions $\mathcal{A}[\![a]\!] : \mathbb{M} \to \mathbb{Z}$, $\mathcal{B}[\![b]\!] : \mathbb{M} \to \mathbb{B}\ (= \{\mathit{true}, \mathit{false}\})$, $\mathcal{P}[\![p]\!] : \mathbb{M} \to \Sigma$, and $\mathcal{C}[\![c]\!] : \mathbb{M} \to \Sigma^*$. We write $\bullet$ for string concatenations, i.e., $\overset{n}{\underset{k=1}{\bullet}} A_k = A_1 \cdots A_n$. Semantics of holes is undefined.

Synthesis Problem Suppose we are given a set of example patterns $\mathcal{E} \subset \Sigma^*$. Then, our goal is to find a complete program in the DSL that correctly describes the given patterns. Formally, we aim to find the body c of the outer loop such that

$$\forall e \in \mathcal{E}.\ \overset{m(N)}{\underset{k=1}{\bullet}} \Big(\big(\mathcal{C}[\![c]\!](m[i \mapsto k])\big)\hookleftarrow\Big) = e$$

where $m = [N \mapsto row_e]$ and row_e is the number of rows (i.e., the number of $\hookleftarrow$ s) in each example e.

4 Synthesis Algorithm

Now we present our algorithm for efficiently solving the synthesis problem, which combines enumerative search, constraint solving, and static program analysis.

Algorithm 1 Synthesis Algorithm

Input: A set of examples $\mathcal{E}$
Output: A program consistent with $\mathcal{E}$

```
1: W ← {s_0} where s_0 = □
2: Γ ← a candidate set of integer triplets
3: repeat
4:     Pick the smallest state s from W
5:     if s is a terminal state then
6:         if solution(s) then return (for i in N do (s; ↩))
7:     else
8:         W ← W ∪ next(s)
9: until W = ∅
```

4.1 Enumerative Search

Enumerative search has been used as one of the most effective ways in program synthesis [Udupa *et al.*, 2013; Feser *et al.*, 2015; Lee *et al.*, 2016; So and Oh, 2017; Feng *et al.*, 2017]. Below, we transform the synthesis problem to a state-search problem and describe a basic enumerative algorithm for it.

Search Problem Suppose a synthesis task $\mathcal{E}$ is given. We define a 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 final solution states.

- States: A state $s \in S$ is a column-loop statement c that possibly has holes ($\diamondsuit$, $\triangle$, $\bigcirc$, $\square$).
- Initial State: An initial state s_0 is a column-loop hole $\square$.
- Transition Relation: Transition relation $(\rightsquigarrow) \subseteq S \times S$ determines next states directly reachable from the current state. Figure 3 shows the relation as a set of inference rules, where $\rightsquigarrow_a$, $\rightsquigarrow_b$, $\rightsquigarrow_p$, and $\rightsquigarrow$ denote the one-step transition of arithmetic expressions, boolean expressions, symbols, and column-loops, respectively. That is, holes are replaced with any other syntactic components of the same type. Given a state s, we write $\mathsf{next}(s)$ for the set of all immediate following states from s, i.e., $\mathsf{next}(s) = \{s' \mid s \rightsquigarrow s'\}$.
- Solution States: Let us write $s \not\rightsquigarrow$ to denote a terminal state, i.e., a state without holes. Then, a state s is a solution of the synthesis problem iff s is a terminal state and it is consistent with all of the given examples:

$$\mathsf{solution}(s) \iff s \not\rightsquigarrow\ \wedge \forall e \in \mathcal{E}.\ \overset{m(N)}{\underset{k=1}{\bullet}} \Big(\big(\mathcal{C}[\![s]\!](m[i \mapsto k])\big)\hookleftarrow\Big) = e.$$

In the transition relation for arithmetic holes

$$\frac{}{\diamondsuit \rightsquigarrow_a x * N + y * i + z}\ (x, y, z) \in \Gamma$$

we assume that a finite set $\Gamma \subset \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$ of integer triples is given at the moment, which will be described in Section 4.2. Note that, with this assumption, the successors (i.e., $\mathsf{next}(s)$) of a state s are always finite.

Search Algorithm Algorithm 1 shows the basic architecture of our synthesis algorithm. The algorithm initializes the workset W with s_0 (line 1). Then, at line 2, we compute the set of integer triples Γ that will be used in programs (Section 4.2). We pick and remove the smallest sized state s from

$$\frac{a \rightsquigarrow_a a'}{j = a \rightsquigarrow_b j = a'} \quad \frac{b_1 \rightsquigarrow_b b_1'}{b_1 \,||\, b_2 \rightsquigarrow_b b_1' \,||\, b_2} \quad \frac{b_2 \rightsquigarrow_b b_2'}{b_1 \,||\, b_2 \rightsquigarrow_b b_1 \,||\, b_2'} \quad \frac{b \rightsquigarrow_b b'}{\mathit{if}\, b\, l_1\, l_2 \rightsquigarrow_p \mathit{if}\, b'\, l_1\, l_2} \quad \frac{a \rightsquigarrow_a a'}{\mathsf{for}\ j\ \mathsf{in}\ a\ \mathsf{do}\ p \rightsquigarrow \mathsf{for}\ j\ \mathsf{in}\ a'\ \mathsf{do}\ p}$$

$$\frac{p \rightsquigarrow_p p'}{\mathsf{for}\ j\ \mathsf{in}\ a\ \mathsf{do}\ p \rightsquigarrow \mathsf{for}\ j\ \mathsf{in}\ a\ \mathsf{do}\ p'} \quad \frac{c_1 \rightsquigarrow c_1'}{c_1; c_2 \rightsquigarrow c_1'; c_2} \quad \frac{c_2 \rightsquigarrow c_2'}{c_1; c_2 \rightsquigarrow c_1; c_2'} \quad \frac{}{\diamondsuit \rightsquigarrow_a x * N + y * i + z}\ (x,y,z) \in \Gamma \quad \frac{}{\triangle \rightsquigarrow_b i = 1}$$

$$\frac{}{\triangle \rightsquigarrow_b i = N} \quad \frac{}{\triangle \rightsquigarrow_b j = \diamondsuit} \quad \frac{}{\triangle \rightsquigarrow_b \triangle \,||\, \triangle} \quad \frac{}{\bigcirc \rightsquigarrow_p l} \quad \frac{}{\bigcirc \rightsquigarrow_p \mathit{if}\, \triangle\, l_1\, l_2}\ (l_1 \neq l_2) \quad \frac{}{\square \rightsquigarrow \mathsf{for}\ j\ \mathsf{in}\ \diamondsuit\ \mathsf{do}\ \bigcirc} \quad \frac{}{\square \rightsquigarrow \square; \square}$$

Figure 3: Transition relation ($(x, y, z) \in \Gamma$, $l \in \{\bigstar, \_\}$)

W (line 4), where the sizes are estimated by our heuristic cost model based on the principle of Occam's razor. If the chosen state is a solution state, we return a program where a body c is that state (line 6). Otherwise, we obtain next states of s, add them to the workset (line 8), and then repeat the loop.

A standard way of improving enumerative search is to prune away syntactically different but semantically equivalent program states (e.g., [Feser *et al.*, 2015; Lee *et al.*, 2016; So and Oh, 2017]). We use three techniques to do so.

First, we exclude states that have unsatisfiable (i.e., always false) expressions, by checking boolean satisfiability. For example, consider a state below:

$$\big(\square; \mathsf{for}\ j\ \mathsf{in}\ N - i\ \mathsf{do}\ \{\mathit{if}\,(j = N - i + 1)\ \bigstar\ \_\}\big).$$

Note that the condition $(j = N - i + 1)$ is unsatisfiable since the enclosing loop iterates only $N - i$ times (i.e., $N - i < N - i + 1$). Pruning the above state is safe, due to the enumerative search for a simpler and semantically equivalent alternative ($\square$; for j in $N - i$ do $\_$). Secondly, we simplify a state that has syntactically the same boolean expressions. For example, $(i = 1 \,||\, i = 1 \,||\, \triangle)$ is simplified to $(i = 1 \,||\, \triangle)$. Lastly, we reorder boolean expressions in a particular order. For instance, assuming $(i = -)$ precedes $(j = -)$ in the order, $(j = 1 \,||\, i = 1)$ is rewritten as $(i = 1 \,||\, j = 1)$. These three optimizations are implicitly conducted by $\mathsf{next}(s)$.

4.2 Constraint Solving for Inferring Integers

A key feature of our algorithm is that the set $\Gamma \subset \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$ of integer component triples is automatically inferred from example patterns. Achieving this consists of two steps. First, we generate constraints that encode necessary conditions for solutions. Secondly, we solve the constraints using an off-the-shelf SMT solver to obtain a model (i.e., Γ).

Specifically, given a set $\mathcal{E}$ of examples, we generate formula $\Phi_{\mathcal{E}} = \bigwedge_{e \in \mathcal{E}} (\phi_e \wedge \varphi_e)$, where ϕ_e and φ_e are generated as follows:

1. ϕ_e constrains the number of iterations of column-loops. Note that every column-loop in a solution program cannot iterate beyond the columns of each given example e. Also, a column-loop iterates at least zero times. Combining these two conditions, we generate

$$\phi_e = \bigwedge_{i=1}^{row_e} 0 \leq x * row_e + y * i + z \leq col_e^i$$

where col_e^i denotes the number of columns (up to the position right before a newline) at ith row in an example $e \in \mathcal{E}$. The linear expression in ϕ_e is an arithmetic expression a (Figure 1) where N is replaced with row_e.

2. φ_e limits the values of z so that the absolute value of z is smaller than the number of rows in each example:

$$\varphi_e = -row_e < z < row_e.$$

The second condition φ_e is intended to rewrite a large value of z by a small one with additional variables (N or i). We do this rewriting because programs with smaller constant terms are likely to generalize better than those with larger constant terms do. We can still find solution programs even with this extra condition, if we can represent an expression with a large z ($\geq |row_e|$) by expressions with small zs ($< |row_e|$). For example, supposing the number of rows of a given example is 5 (i.e., $row_e = 5$), we can rewrite an arithmetic expression 6 by $N + 1$.

Note that solving the constraint $\Phi_{\mathcal{E}}$ always produces a finite number of integer triples Γ. To see why, first observe that $\Phi_{\mathcal{E}}$ has a series of inequalities over x, y, and z, connected by conjunctions. Then, we need to show that any two of x, y, z are finitely bounded through the inequalities: in our case, y and z. We explain y-case first. Assuming the number of rows in each example is at least two (i.e., $\forall e \in \mathcal{E}.row_e \geq 2$), we can conclude the range of y is finite from ϕ_e, because only the coefficients of ys vary in each conjunct of ϕ_e, and each conjunct in ϕ_e has both a lower- and upper bound (0 and col_e^i, respectively). For example, from ϕ_e in Example 1 (i.e., the first four conjuncts), we get $-1 \leq y \leq 2$. Secondly, from φ_e, we can easily see that z also has a finite range.

Example 1 *Consider the Problem 1 in Section 2. We can generate constraints that correspond to the given example:*

$$0 \leq 4 * x + 1 * y + z \leq 4 \wedge 0 \leq 4 * x + 2 * y + z \leq 5 \wedge$$
$$0 \leq 4 * x + 3 * y + z \leq 6 \wedge 0 \leq 4 * x + 4 * y + z \leq 7 \wedge$$
$$-4 < z < 4$$

where $row_e = 4$, and col_e^1 through col_e^4 are 4,5,6,7, respectively. Solving the constraints, we get $\Gamma = \{$ (-1, 1, 3), (-1, 2, 2), (-1, 2, 3), (0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 0, 3), (0, 1, -1), (0, 1, 0), (0, 1, 1), (0, 1, 2), (0, 1, 3), (0, 2, -2), (0, 2, -1), (1, -1, 0), (1, -1, 1), (1, 0, -3), (1, 0, -2), (1, 0, -1), (1, 0, 0), (1, 1, -3), (1, 1, -2), (1, 1, -1), (2, -1, -3) $\}$. Observe that (1,-1,0) and (0,2,-1) appears in the solution program.

4.3 Pruning using Static Program Analysis

We observed that many candidates encountered during the enumerative search can never reach solution states, only hindering search for solutions. A common example is given in Example 2. Our goal is to effectively prune away this type of candidates in a safe manner. To achieve the goal, we design a static analysis, which enables us to identify states that eventually fail to be solutions, and perform pruning using it.

$$\widehat{\mathcal{A}}[\![x*N+y*i+z]\!](\widehat{m}) = x\widehat{*}\widehat{m}(N)\ \widehat{+}\ y\widehat{*}\widehat{m}(i)\ \widehat{+}\ z$$
$$\widehat{\mathcal{B}}[\![i=1]\!](\widehat{m}) = \widehat{m}(i)\ \widehat{=}\ 1$$
$$\widehat{\mathcal{B}}[\![i=N]\!](\widehat{m}) = \widehat{m}(i)\ \widehat{=}\ \widehat{m}(N)$$
$$\widehat{\mathcal{B}}[\![j=a]\!](\widehat{m}) = \widehat{m}(j)\ \widehat{=}\ \widehat{\mathcal{A}}[\![a]\!](\widehat{m})$$
$$\widehat{\mathcal{B}}[\![b_1 \mid\mid b_2]\!](\widehat{m}) = \widehat{\mathcal{B}}[\![b_1]\!](\widehat{m})\ \widehat{\vee}\ \widehat{\mathcal{B}}[\![b_2]\!](\widehat{m})$$
$$\widehat{\mathcal{P}}[\![l]\!](\widehat{m}) = \{l\}$$
$$\widehat{\mathcal{P}}[\![\mathit{if}\ b\ l_1\ l_2]\!](\widehat{m}) = \begin{cases} \{l_1\}\cup\{l_2\} & (\widehat{\mathcal{B}}[\![b]\!](\widehat{m}) = \top_{\widehat{\mathbb{B}}}) \\ \{l_1\} & (\widehat{\mathcal{B}}[\![b]\!](\widehat{m}) = \widehat{true}) \\ \{l_2\} & (\widehat{\mathcal{B}}[\![b]\!](\widehat{m}) = \widehat{false}) \end{cases}$$
$$\widehat{\mathcal{C}}[\![\mathsf{for}\ j\ \mathsf{in}\ a\ \mathsf{do}\ p]\!](\widehat{m}) = \begin{cases} (\widehat{\mathcal{P}}[\![p]\!](\widehat{m}[j\mapsto \top_{\widehat{\mathbb{A}}}]))^* & (\widehat{\mathcal{A}}[\![a]\!](\widehat{m}) = \top_{\widehat{\mathbb{A}}}) \\ \overset{\widehat{\mathcal{A}}[\![a]\!](\widehat{m})}{\underset{k=1}{\bullet}} \widehat{\mathcal{P}}[\![p]\!](\widehat{m}[j\mapsto k]) & \text{otherwise} \end{cases}$$
$$\widehat{\mathcal{C}}[\![c_1; c_2]\!](\widehat{m}) = \big(\widehat{\mathcal{C}}[\![c_1]\!](\widehat{m})\big)\big(\widehat{\mathcal{C}}[\![c_2]\!](\widehat{m})\big)$$

Figure 4: Abstract semantics

Designing analysis In order to identify failure candidates, we first define abstract semantic functions in Figure 4: $\widehat{\mathcal{A}}[\![a]\!] : \widehat{\mathbb{M}} \to \widehat{\mathbb{Z}}$, $\widehat{\mathcal{B}}[\![b]\!] : \widehat{\mathbb{M}} \to \widehat{\mathbb{B}}$, $\widehat{\mathcal{P}}[\![p]\!] : \widehat{\mathbb{M}} \to 2^{\Sigma}$, and $\widehat{\mathcal{C}}[\![c]\!] : \widehat{\mathbb{M}} \to 2^{\Sigma^*}$. An abstract memory state $\widehat{m} \in \widehat{\mathbb{M}}$ is a mapping from variables to abstract integers ($\widehat{\mathbb{Z}}$), where $\widehat{\mathbb{Z}} = \{\top_{\widehat{\mathbb{A}}}\} \cup \mathbb{Z}$ with a partial order $z_1 \sqsubseteq_{\widehat{\mathbb{Z}}} z_2 \iff (z_2 = \top_{\widehat{\mathbb{A}}})$. $\widehat{\mathbb{B}} = \{\top_{\widehat{\mathbb{B}}}, \widehat{true}, \widehat{false}\}$ is abstract boolean values with a partial order $b_1 \sqsubseteq_{\widehat{\mathbb{B}}} b_2 \iff (b_2 = \top_{\widehat{\mathbb{B}}})$.

A key component in designing the analysis is to define *sound* semantics for holes, in order to over-approximate all possible concrete behaviors of terminal states reachable from a current candidate state. For the arithmetic- and boolean expressions holes ($\diamondsuit$, $\triangle$), we conservatively assign top values, i.e., $\widehat{\mathcal{A}}[\![\diamondsuit]\!](\widehat{m}) = \top_{\widehat{\mathbb{A}}}$ and $\widehat{\mathcal{B}}[\![\triangle]\!](\widehat{m}) = \top_{\widehat{\mathbb{B}}}$. For the symbol hole ($\bigcirc$), we take the union of $\bigstar$ and $\_$, i.e., $\widehat{\mathcal{P}}[\![\bigcirc]\!](\widehat{m}) = \{\bigstar\}\cup\{\_\}$. Lastly, we compute the column-loop hole ($\square$) as Kleene Star on star and blank, i.e., $\widehat{\mathcal{C}}[\![\square]\!] = \big(\{\bigstar\} \cup \{\_\}\big)^*$. A novel aspect is that we use regex operations for over-approximations of string values.

Pruning with analysis Suppose a set of examples $\mathcal{E} \subset \Sigma^*$ is given, and we are exploring a candiate s. Then, we prune the candidate if the following predicate $\mathsf{pruned}(s)$ holds:

$$\mathsf{pruned}(s) \iff \exists e \in \mathcal{E}.e \notin \overset{\widehat{m}(N)}{\underset{k=1}{\bullet}} \Big(\big(\widehat{\mathcal{C}}[\![c]\!](\widehat{m}[i\mapsto k])\big)\{\hookleftarrow\}\Big)$$

where $\widehat{m} = [N \mapsto row_e]$. That is, we prune the candidate iff the over-approximated result obtained by running the analysis on the candidate does not contain the desired output (i.e., the given example) for some examples $e \in \mathcal{E}$, since all further search of such the candidate will fail to produce solutions.

Example 2 *Suppose the pattern example of Problem 1 in Section 2 is given. Then, a candidate* $\big((\mathsf{for}\ j\ \mathsf{in}\ i\ \mathsf{do}\ \bigstar); \square\big)$ *needs not be searched further, because the analysis result on the candidate (the right below, starting with* $\bigstar$*) does not contain the expected output (the left below, starting with* $\_$*):*

$$\_\_\_\bigstar\hookleftarrow\cdots \quad \notin \quad \bigstar(\{\bigstar\} \cup \{\_\})^*\hookleftarrow\cdots$$

We finalize our synthesis algorithm by revising $\mathsf{next}(s)$:

$$\mathsf{next}(s) = \textbf{if}\ \mathsf{pruned}(s)\ \textbf{then}\ \emptyset\ \textbf{else}\ \{s' \mid s \rightsquigarrow s'\}$$

5 Evaluation

Experimental setup To demonstrate the effectiveness of our approach, we collected 40 benchmarks from several online forums. These problems are comprised of various geometric patterns that are helpful for students' learning, including challenging ones. All of the experiments were conducted on MacBook Pro with Intel Core i7 and 16GB of memory.

Algorithm performance The experimental results show that our synthesis algorithm is remarkably efficient; PAT solves the benchmarks 1.2 seconds on average, only from 1–3 examples (the column "Final" and "Ex" in Table 1). However, when we performed only the enumerative search (Section 4.1, 4.2) without our program-analysis-guided pruning (Section 4.3), seven of the benchmarks timed out ($>$ 1 hour), as shown in the column "Enum" in Table 1. Assuming the running time of each of the seven benchmark as 1 hour, the average running time becomes 737.9 seconds (x615 slowdown). This result indicates two things. First, although our DSL is restrictive to introductory pattern programs, its search space is never trivial, hence an effective pruning method should be applied. Secondly, our pruning is powerful to dramatically cut down on the search space.

After the experiments, we manually checked that all of the synthesized programs are correct. Also, as demonstrated in Section 2, the generated programs are explainable, understandable and instructive, capturing core pattern rules of the given pattern examples.

User study We also evaluated PAT with 23 undergraduates, who have taken the introductory programming course, in order to inspect whether PAT is actually helpful to students. The study proceeded as follows. We first requested the students to solve several pattern programming problems, and submit their code in the C language. After receiving the code, we asked the students to freely use PAT in our web demo page for one day. Finally, we asked four survey questions:

Q1. Programs generated by PAT are overall simple and easy to understand.

Q2. Compared to their own submitted programs, programs generated by PAT are simpler and easier to understand.

Q3. PAT helps to learn programming.

Q4. PAT is easy to use.

These questions were designed to evaluate code qualities of PAT (Q1, Q2), helpfulness in learning programming (Q3), and usability (Q4). Note that Q2 is a rigorous version of Q1, in that the students can answer to Q1 just by running PAT for certain patterns and observing generated solutions, without efforts to solve problems. Especially for Q3, we requested to describe reasons for their answers, allowing multiple responses. In the demo page, programs written by PAT were displayed after converted into C from our DSL.

Table 2 shows results of our survey. For Q1 and Q2, the students highly appreciated both code simplicity and readability of PAT. Notably, for Q2, 91% responded that PAT's programs have higher or similar qualities, compared to code written by themselves. We believe that reasons for this result may come from our simple DSL design and our search strategy, which prefers a small-sized candidate first (Section 4.1).

No	Description	Ex	Time (sec)	
			Final	Enum
1	Square	2	0.0	0.0
2	Hollow square	2	0.1	3.2
3	Parallelogram	2	0.0	0.1
4	Hollow parallelogram	2	0.5	$\perp$
5	Mirrored parallelogram	2	0.0	0.1
6	Hollow mirrored parallelogram	2	0.6	2724.4
7	Right triangle	2	0.0	0.0
8	Hollow right triangle	3	0.0	0.1
9	Mirrored right triangle	2	0.0	0.0
10	Hollow mirrored right triangle	3	0.1	0.5
11	Inverted right triangle	2	0.0	0.0
12	Hollow inverted right triangle	2	0.0	0.0
13	Inverted mirrored right triangle	2	0.0	0.0
14	Hollow inverted mirrored right triangle	2	0.1	0.6
15	Isosceles triangle	1	0.0	0.2
16	Hollow isosceles triangle	2	0.1	2.4
17	Invereted isosceles triangle	2	0.0	0.1
18	Hollow inverted isosceles triangle	1	0.3	5.6
19	Rectangle with an empty trapezoid	2	1.0	298.9
20	Inverted rectangle with an empty trapezoid	2	1.2	610.5
21	Obtuse triangle	2	0.0	0.1
22	Hollow obtuse triangle	2	0.3	5.7
23	Mirrored obtuse triangle	2	0.0	0.0
24	Hollow mirrored obtuse triangle	2	0.1	3.4
25	Inverted obtuse triangle	1	0.0	0.1
26	Hollow inverted obtuse triangle	2	0.2	4.6
27	Inverted mirrored obtuse triangle	2	0.0	0.0
28	Hollow inverted mirrored obtuse triangle	2	0.2	6.3
29	V-pattern	2	0.0	0.4
30	Trapezoid	2	0.1	0.9
31	Hollow trapezoid	2	2.4	$\perp$
32	Inverted trapezoid	2	0.1	0.8
33	Hollow inverted trapezoid	2	3.0	$\perp$
34	Combination of #4 and #10	3	24.8	$\perp$
35	Combination of #6 and #8	2	1.0	$\perp$
36	Hollow square with diagonals	2	5.2	$\perp$
37	N-pattern	2	0.4	73.2
38	Inclined N-pattern	2	0.6	186.9
39	W-pattern	2	4.2	$\perp$
40	Bow-tie pattern	2	0.6	385.3
	Average		1.2	> 737.9

Table 1: Performance of PAT. $\perp$ denotes timeout ($>$ 1 hour).

	Agree	Neutral	Disagree
Q1	87% (20)	9% (2)	4% (1)
Q2	52% (12)	39% (9)	9% (2)
Q3	74% (17)	13% (3)	13% (3)
Q4	78% (18)	22% (5)	0% (0)

Table 2: Responses from users. (n) shows the actual number.

In particular, for the benchmark #39, our manual study on the 10 correct submissions reveals that, only two successfully captured core pattern rules. The remaining eight were composed of messy rules without core rules, although they are somehow correct; the worst case solution consisted of 54 lines in C with 7 inner loops (vs. 18 lines with one inner loop by PAT). For Q3, 74% agreed that PAT is helpful in learning programming. For Q4, no negative answers were reported.

We further examined reasons behind students' answers for Q3. 74% (17 students) answered "Agree" for following reasons: improving programming skills by learning instructive code generated by PAT (71%, 12/17), realtime-feedback without asking to human instructors (65%, 11/17), and role as an automated teacher for shy students who are reluctant to ask in classrooms (47%, 8/17).

In contrast, among the three students who disagreed with Q3, one expressed that PAT's code are hard to follow, which conflicts with the majority. The rest two disagreed, worrying about students' blind reliance on PAT without enough endeavors to solve problems. However, we still believe that PAT can be helpful under proper guidance of instructors, e.g., allowing to use PAT only after submitting assignments.

Finally, we collected additional feedback for future improvements: translations into other languages (e.g., Java), and dealing with more complex patterns. The former can be easily done. For the latter, we believe that extending PAT to solve programming-competition-level pattern problems, which require to write complicated recursive functions with nontrivial arguments [3], is one interesting research direction.

6 Related Work

Automatic programming tutor There have been a number of researches on building intelligent programming tutor systems that aim to help novice programmers: generating feedback on semantic errors in students' submissions [Kim *et al.*, 2016; Singh *et al.*, 2013; Gulwani *et al.*, 2014; Kaleeswaran *et al.*, 2016], step-by-step hint generation for Haskell [Gerdes *et al.*, 2017] and Python [Rivers and Koedinger, 2017], and many others. Our work differs from prior works in two ways. First, PAT can help students who do not know how to start writing programs by synthesizing solutions from examples, while prior works aim at producing feedback on code already written by students. Secondly, to the best of our knowledge, PAT is the first automatic tutor that helps to practice pattern programs.

Comparisons with other synthesis approaches In many program synthesis applications, the DSL has played a key role in speeding up search process by reasonably restricting search space on target domains: automating string manipulation in spreadsheet [Gulwani, 2011; Kini and Gulwani, 2015; Raza *et al.*, 2015; Wu and Knoblock, 2015; Ellis and Gulwani, 2017], automated data extraction [Raza and Gulwani, 2017], and transformations on tree data structures [Yaghmazadeh *et al.*, 2016], etc. In this context, we designed the DSL for efficiently synthesizing pattern programs.

Technically, our algorithm differs from previous works in two aspects. First, most notably, by applying constraint solving, PAT can automatically infer integer components, which will be used in a solution program, without manual provision from end-users. By easing burden on users in this way, we believe that our idea of applying constraint solving for inferring program components can benefit many other program synthesizers, e.g., [Wang *et al.*, 2017; So and Oh, 2017]. Secondly, we developed an effective and safe pruning method that performs static program analysis using regex operations.

7 Conclusion

In this paper, we presented a new programming-by-example system that synthesizes pattern programs from examples. We first carefully designed the DSL for efficiently synthesizing a variety of pattern programs. Next, we developed a novel synthesis algorithm that combines enumerative search, constraint solving, and program analysis. We demonstrated the effectiveness of our approach with 40 pattern tasks, and the helpfulness with 23 students.

[3] e.g., https://www.acmicpc.net/problem/2447

Acknowledgments

We thank Joonho Lee and Dowon Song for discussions on designing the survey questions. This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1701-09. 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

[Ellis and Gulwani, 2017] Kevin Ellis and Sumit Gulwani. Learning to learn programs from examples: Going beyond program structure. In *Proceedings of International Joint Conference on Artificial Intelligence, IJCAI*, pages 1638–1645, 2017.

[Feng *et al.*, 2017] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 422–436, 2017.

[Feser *et al.*, 2015] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 229–239, 2015.

[Gerdes *et al.*, 2017] Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and L. Thomas van Binsbergen. Ask-elle: an adaptable programming tutor for haskell giving automated feedback. *I. J. Artificial Intelligence in Education*, 27(1):65–100, 2017.

[Gulwani *et al.*, 2014] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. Feedback generation for performance problems in introductory programming assignments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE, pages 41–51, 2014.

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

[Gulwani, 2011] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 317–330, 2011.

[Kaleeswaran *et al.*, 2016] Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. Semi-supervised verified feedback generation. In *Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE, pages 739–750, 2016.

[Kim *et al.*, 2016] Dohyeong Kim, Yonghwi Kwon, Peng Liu, I. Luk Kim, David Mitchel Perry, Xiangyu Zhang, and Gustavo Rodriguez-Rivera. Apex: Automatic programming assignment error explanation. In *Proceedings of ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, pages 311–327, 2016.

[Kini and Gulwani, 2015] Dileep Kini and Sumit Gulwani. Flashnormalize: Programming by examples for text normalization. In *Proceedings of International Conference on Artificial Intelligence*, IJCAI, pages 776–783, 2015.

[Lee *et al.*, 2016] Mina Lee, Sunbeom So, and Hakjoo Oh. Synthesizing regular expressions from examples for introductory automata assignments. In *Proceedings of ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE, pages 70–80, 2016.

[Raza and Gulwani, 2017] Mohammad Raza and Sumit Gulwani. Automated data extraction using predictive program synthesis. In *Proceedings of AAAI Conference on Artificial Intelligence*, pages 882–890, 2017.

[Raza *et al.*, 2015] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. Compositional program synthesis from natural language and examples. In *Proceedings of International Conference on Artificial Intelligence*, IJCAI, pages 792–800, 2015.

[Rivers and Koedinger, 2017] Kelly Rivers and Kenneth R. Koedinger. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *I. J. Artificial Intelligence in Education*, 27(1):37–64, 2017.

[Singh *et al.*, 2013] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 15–26, 2013.

[So and Oh, 2017] Sunbeom So and Hakjoo Oh. Synthesizing imperative programs from examples guided by static analysis. In *Static Analysis - International Symposium, SAS*, pages 364–381, 2017.

[Udupa *et al.*, 2013] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. Transit: Specifying protocols with concolic snippets. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 287–296, 2013.

[Wang *et al.*, 2017] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive sql queries from input-output examples. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 452–466, 2017.

[Wu and Knoblock, 2015] Bo Wu and Craig A. Knoblock. An iterative approach to synthesize data transformation programs. In *Proceedings of International Conference on Artificial Intelligence*, IJCAI, pages 1726–1732, 2015.

[Yaghmazadeh *et al.*, 2016] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. Synthesizing transformations on hierarchically structured data. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 508–521, 2016.