Inductive Synthesis of Structurally Recursive Functional Programs from Non-recursive Expressions

WOOSUK LEE*, Hanyang University, South Korea
HANGYEOL CHO, Hanyang University, South Korea

We present a novel approach to synthesizing recursive functional programs from input-output examples. Synthesizing a recursive function is challenging because recursive subexpressions should be constructed while the target function has not been fully defined yet. We address this challenge by using a new technique we call block-based pruning. A block refers to a recursion- and conditional-free expression (i.e., straight-line code) that yields an output from a particular input. We first synthesize as many blocks as possible for each input-output example, and then we explore the space of recursive programs, pruning candidates that are inconsistent with the blocks. Our method is based on an efficient version space learning, thereby effectively dealing with a possibly enormous number of blocks. In addition, we present a method that uses sampled input-output behaviors of library functions to enable a goal-directed search for a recursive program using the library. We have implemented our approach in a system called Trio and evaluated it on synthesis tasks from prior work and on new tasks. Our experiments show that Trio outperforms prior work by synthesizing a solution to 98% of the benchmarks in our benchmark suite.

CCS Concepts: • **Software and its engineering** → **Programming by example**; **Automatic programming**.

Additional Key Words and Phrases: Programming by Example, Recursive Functional Programs, Synthesis

ACM Reference Format:
Woosuk Lee and Hangyeol Cho. 2023. Inductive Synthesis of Structurally Recursive Functional Programs from Non-recursive Expressions. *Proc. ACM Program. Lang.* 7, POPL, Article 70 (January 2023), 31 pages. https://doi.org/10.1145/3571263

1 INTRODUCTION

Recent years have witnessed a surge of interest in recursive functional program synthesis [Albarghouthi et al. 2013; Farzan and Nicolet 2021; Feser et al. 2015; Kneuss et al. 2013; Lubin et al. 2020; Miltner et al. 2022; Osera and Zdancewic 2015; Polikarpova et al. 2016]. In particular, because input-output examples are readily available, inductive synthesis of recursive functional programs has gained a lot of attention, witnessing significant strides. Inductive synthesis problems are typically expressed as a combination of algebraic data types, a library of *external* operators over the data types, and input-output examples that should be satisfied by the target function to be synthesized.

Despite recent advances, synthesizing recursive functional programs from input-output examples is still challenging, mainly due to the following two factors.

[*] Corresponding author.

Authors' addresses: Woosuk Lee, Hanyang University, Dept. of Computer Science & Engineering, South Korea, woosuk@hanyang.ac.kr; Hangyeol Cho, Hanyang University, Dept. of Computer Science & Engineering (major in Bio Artificial Intelligence), South Korea, pigon8@hanyang.ac.kr.

2475-1421/2023/1-ART70
https://doi.org/10.1145/3571263

- *Recursive calls*: recursive data types often necessitate recursive calls, which are nontrivial to synthesize. That is because we should be able to reason about the target function yet to be defined during the search. As a workaround, previous approaches [Albarghouthi et al. 2013; Osera and Zdancewic 2015] require the user to provide a *trace-complete specification* where the behaviors of recursive call expressions are part of the specification [1]. However, writing a trace-complete specification is quite unintuitive and difficult even for experts who are familiar with the synthesizers. To overcome this limitation, there have been previous methods, including specification strengthening [Miltner et al. 2022], partial evaluation, and constraint solving [Lubin et al. 2020]. However, these approaches have their weaknesses, occasionally suffering from scalability issues even for small programs (see Section 6).
- *External operators*: to synthesize programs that utilize various operators over algebraic data types, synthesizers often require the user to provide a library of external operators. However, there is no general method for accelerating synthesis by exploiting the semantics of such external operators. Previous methods (e.g., [Feser et al. 2015]) rely on pre-defined deductive rules only applicable to a fixed set of combinators (e.g., `map`, `fold`) or resort to naive enumeration. Therefore, the scalability issues worsen in the presence of a targeted library of external operators.

In this paper, we propose a novel approach to the inductive synthesis of recursive functional programs that addresses these challenges.

Our method for handling recursion, which we call *block-based pruning*, is to carry out synthesis in two phases: (1) synthesis of *blocks* satisfying the given examples followed by (2) synthesis of a recursive program. We define a *block* as a recursion- and conditional-free expression (i.e., straight-line program) that yields an output for a particular input, which is called *trace* in the prior work [Kitzelmann and Schmid 2006; Summers 1986]. For each input-output example, we first synthesize as many blocks satisfying that example as possible. Based on an efficient version space learning, we effectively deal with possibly an enormous number of blocks [2]. And then, we explore the space of recursive programs top-down, generating incomplete candidate programs with holes (which we call *hypotheses*). For each hypothesis containing recursive calls, we transform it into blocks possibly with holes (which we call *open blocks*) by symbolic evaluation interleaved with concrete evaluation. If the open blocks cannot be the blocks synthesized in the earlier phase by filling the holes, the hypothesis is determined to be *inconsistent* with the blocks and is discarded.

Our method for handling external operators, which we call *library sampling*, is to sample input-output behaviors of library functions and use them for synthesis. This method enables a divide-and-conquer strategy called *top-down propagation* (or top-down deductive search) for synthesizing expressions that call *arbitrary* external operators. Top-down propagation hypothesizes an overall structure of the desired program satisfying given input-output examples and then performs deductive reasoning to recursively deduce new examples that should be satisfied by missing subexpressions. For example, suppose we want to synthesize a list manipulating program satisfying an input-output example $[1, 2] \mapsto [3, 4]$. After hypothesizing that the desired program is of form $\mathtt{map}(f, x)$ where x denotes input and f is an unknown function subexpression to be synthesized, we can generate two new input-output examples for f as a synthesis subproblem: $1 \mapsto 3$ and $2 \mapsto 4$. This process is recursively repeated until all subproblems are solved. When hypothesizing the desired program is a function call expression involving external operators, we can use the

[1] For example, suppose the user tries to provide input-output examples $[] \mapsto 0$ and $[1, 2, 3] \mapsto 3$ to synthesize a function that returns the length of an integer list. To make the specification trace-complete, the user should also provide two additional input-output examples: $[2, 3] \mapsto 2$ and $[3] \mapsto 1$.

[2] E.g., a graph of 2470 nodes is used to represent over 7 million blocks (for the `list_rev_append` benchmark in Section 6).

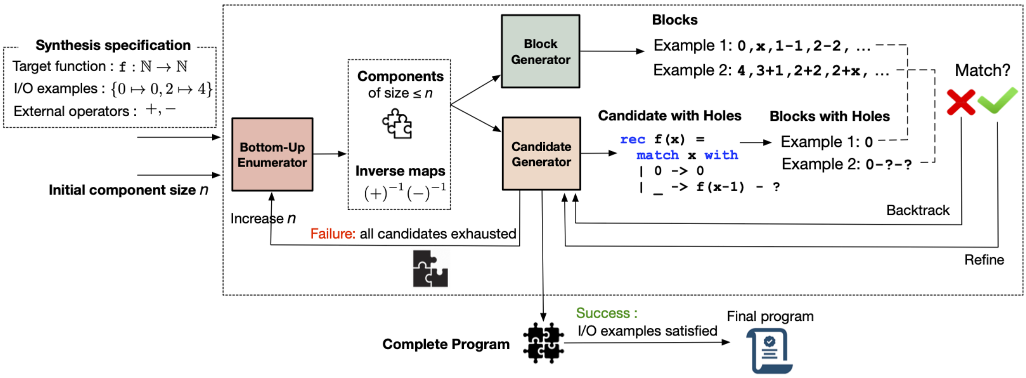

Fig. 1. High-level architecture of our synthesis algorithm.

input-output samples of the library functions to deduce new examples for missing sub-expressions. This method is applicable even for black-box libraries.

Fig. 1 presents the overall architecture of our synthesis algorithm, inspired by a recently proposed synthesis strategy [Lee 2021]. Our synthesis algorithm consists of three key modules, namely *Bottom-up enumerator*, *Block generator*, and *Candidate generator*:

- ***Bottom-up enumerator***: Given synthesis specification comprising input-output examples and usable external operators, and a number n, the *Bottom-up enumerator* module generates two ingredients for the other modules: *components* and *inverse maps*. The components are expressions (of size $\leq n$) that can be used to construct blocks and recursive programs. The inverse maps are finite maps from outputs to inputs of the external operators and derived from input-output samples collected from concrete evaluation of the external operators.
- ***Block generator***: Given the two ingredients from *Bottom-up enumerator*, the *Block generator* module generates blocks. For each input-output example, the module generates as many satisfying blocks as possible using the components. The block generation phase can be quickly done by top-down propagation. To control blocks in an enormous amount, we make use of version space representations to efficiently enumerate and store them.
- ***Candidate generator***: Given the blocks generated by *Block generator*, the *Candidate generator* module searches for a solution also by top-down propagation. Starting with an empty program, it generates a sequence of hypotheses (i.e., partial programs with holes). During the search, any hypothesis inconsistent with the blocks is discarded early. *Candidate generator* keeps generating hypotheses until it finds a solution, or all candidates have been explored in the search space. If a solution cannot be found using the current components, the whole process is repeated with the component size n increased by 1, thereby exploring the larger search space in the next iteration.

Our algorithm eventually finds a solution if it exists because *Bottom-up enumerator* will eventually enumerate a solution of finite size. Also, our method does not require the user to provide unintuitive trace-complete specifications.

We implemented our approach in a tool called Trio. We evaluate Trio on 60 benchmarks: 45 out of 60 are from prior work [Osera and Zdancewic 2015], and the others are newly added. We use two types of specifications: (1) input-output examples and (2) reference implementations. Our evaluation results suggest that Trio is more scalable than prior work on all types of specifications. In particular, our tool can synthesize 98% (59) of the functions from input/output examples and 95% (57) of the functions from reference implementations. We also compare Trio against simpler variants that do not perform either block-based pruning or library sampling, and we empirically prove the efficacy of the two techniques.

Our contributions are as folllows:

- A novel general method for synthesizing recursive programs from input-output examples: We propose a general algorithm for effectively synthesizing recursion- and calls to external operators. We believe our method is potentially applicable to other synthesis contexts.
- Confirming the method's effectiveness in an extensive experimental evaluation: We have conducted an extensive experimental evaluation on synthesis benchmarks from prior work and new benchmarks. Furthermore, we publicly release the implementation of our approach as a tool called Trio (available at https://github.com/pslhy/trio).

2 OVERVIEW

In this section, we give an overview of our method using the problem of synthesizing a recursive function mul for multiplying two natural numbers. The specification for the problem comprises an inductive data type for natural numbers, an external operator add for adding two natural numbers that can be used to synthesize mul, and input-output examples embedded in a *hypothesis*. A hypothesis is a program that may have placeholders for missing expressions. We will call such a placeholder a *hole* (denoted □), which is associated with input-output examples that should be satisfied by a sub-expression in that position. In the following specification (in OCaml-like syntax),

```
type nat = Z | S of nat
rec add (x : nat, y : nat) : nat = match x with Z -> Z | S _ -> S (add (S⁻¹(x), y))
rec mul (x : nat, y : nat) : nat = □ᵢₙ
```

$\square_{in}$ is a hole associated with the set of input-output examples $\{(0,1) \mapsto 0, (1,2) \mapsto 2, (2,1) \mapsto 2\}$. $\mathtt{S}^{-1}$ denotes a *destructor* which extracts the subcomponent of a constructor application of S. Such destructors obviate the need for introducing new variables bound by patterns in match expressions. The following program is a solution.

```
Psol = rec mul (x : nat, y : nat) : nat =
       match x with Z -> Z | S _ -> add (mul (S⁻¹(x), y), y)
```

We will describe how the three modules of our system interact with each other to synthesize the desired program. For brevity, we will often use literals $0, 1, \cdots$ as syntactic sugar for the corresponding naturals $\mathtt{Z}, \mathtt{S(Z)}, \cdots$.

Component Generation and Library Sampling. *Bottom-up enumerator* first generates the following component pool **C** of expressions whose size is not greater than some user-provided upper bound.

$$\mathbf{C} = \{\mathtt{x}, \mathtt{y}, \mathtt{S}^{-1}(\mathtt{x}), \mathtt{S}^{-1}(\mathtt{S}^{-1}(\mathtt{x})), \mathtt{S}^{-1}(\mathtt{y}), \mathtt{S}^{-1}(\mathtt{S}^{-1}(\mathtt{y})), \mathtt{Z}, \mathtt{add}, (\mathtt{S}^{-1}(\mathtt{x}), \mathtt{y}), \cdots\}$$

During bottom-up enumeration, it adopts the existing pruning technique based on *observational equivalence* to avoid maintaining multiple components of the same behaviors with respect to the input examples. [3] This pruning technique drastically reduces the number of components by removing redundant expressions, which leads to overall performance gains. These components will be used to construct blocks and recursive programs in the following phases.

[3] Whenever a new program is enumerated, it is checked if it is "observationally equivalent" to any of the programs already constructed; i.e., those which produce the same outputs on inputs that were given as a specification. If so, the new program is discarded (e.g., $\mathtt{x} + \mathtt{x}$ is discarded if $2 \times \mathtt{x}$ is already enumerated). This is done to avoid enumerating redundant programs.

Next, for each function that a component in **C** may evaluate to, it constructs an inverse map of the function through a method we call library sampling. An inverse map of a function is a finite map from output values to input values of the function. Because add is the only function component, we construct the inverse map of add, which can be derived from input-output samples of add. Such samples can be obtained by evaluating add with input values that are not greater than the values in the examples. The reason behind this choice is that we aim to synthesize *structurally decreasing* recursive programs like previous approaches [Frankle et al. 2016; Lubin et al. 2020; Miltner et al. 2022; Osera and Zdancewic 2015] where arguments of recursive calls are strictly decreasing, and we observe inputs to the target function often flow to the external operators as arguments. Using numbers not greater than the greatest number (2) in the input-output examples (i.e., $(0,0),(0,1),\cdots,(2,2)$) as inputs, we evaluate the add function and obtain the inverse map $\mathsf{add}^{-1} = \{0 \mapsto \{(0,0)\}, 1 \mapsto \{(0,1),(1,0)\}, \cdots, 4 \mapsto \{(2,2)\}\}$.

Block Generation. Next, for each input-output example, *Block generator* generates a set of blocks satisfying that example. Given an input-output example, it adds all the components in **C** satisfying the example to the block set. Irrespective of whether or not any component is added, to find as various blocks as possible, it continues by hypothesizing about the structure of the other possible blocks and deduces new input-output examples that should be satisfied by missing holes. For each hole, it recursively searches for all the blocks that satisfy the hole. For example, consider the second input-output example $(1,2) \mapsto 2$. *Block generator* first finds all components in **C** that satisfy the example. Because the desired output 2 is the value of the second parameter y, y is added to the set of blocks. The search continues by hypothesizing about all possible structures of the other blocks. Suppose it attempts to find blocks of the form $\mathsf{S}\,(\cdots)$, generating a hypothesis $\mathsf{S}(\square_1)$. The hole $\square_1$ is associated with $(1,2) \mapsto 1$ where the output example is obtained by removing the constructor head S from the output example 2. Because the desired output 1 is the value of the first parameter x, `S(x)` is added to the block set. For other possible blocks in place of $\square_1$, it attempts to find blocks of the form $\mathsf{add}\,(\cdots)$. To generate hypotheses involving the external operator, it uses the inverse map of add. Since $\mathsf{add}^{-1}(1) = \{(0,1),(1,0)\}$, it generates two hypotheses $\mathsf{S}(\mathsf{add}\,(\square_2,\square_3))$ and $\mathsf{S}(\mathsf{add}\,(\square_3,\square_2))$ where $\square_2$ and $\square_3$ are associated with $(1,2) \mapsto 0$ and $(1,2) \mapsto 1$, respectively. By finding components satisfying the holes, $\mathsf{S}(\mathsf{add}\,(\mathsf{Z},\mathsf{Z}))$, $\mathsf{S}(\mathsf{add}\,(\mathsf{Z},\mathsf{x}))$, $\mathsf{S}(\mathsf{add}\,(\mathsf{x},\mathsf{Z}))$, and $\mathsf{S}(\mathsf{add}\,(\mathsf{x},\mathsf{x}))$ are added to the block set. It further refines the holes $\square_2$ and $\square_3$ by recursively generating other hypotheses in a similar manner to find more blocks.

During the search, hypotheses containing recursive calls and match expressions are not taken into account because resulting blocks should be recursion- and conditional-free. Let us denote $\mathbf{B}_i$ as the set of blocks for the i-th input-output example. We obtain the following blocks.

$(0,1) \mapsto 0$: $\mathbf{B}_0$ = {0, x, add (0,0), add (0, x), ...}
$(1,2) \mapsto 2$: $\mathbf{B}_1$ = {2, y, S(add (0, x)), add (S^{-1}(x),y), ...}
$(2,1) \mapsto 2$: $\mathbf{B}_2$ = {x, S(y), add (x, Z), add (add (S^{-1}(x), y), y), ...}

Because there are often infinitely many blocks satisfying each example, we limit the maximum number of steps of top-down propagation to ensure the termination of the block generation phase. For example, if we set the maximum number to be 1, in the above example, we would not recursively generate other hypotheses for the holes $\square_2$ and $\square_3$ as we already went through one step of top-down propagation. Even though we finitize the search space, there are often still many blocks. To efficiently enumerate and store them, we use a version space representation, which is a data structure that compactly represents a large set of programs (see Section 4.4).

Candidate Generation. Equipped with the blocks generated by *Block generator*, *Candidate generator* searches for the desired recursive program by performing top-down propagation, similar to what *Block generator* does but with a few differences: recursive calls and match expressions are generated, and all the input-output examples are considered at once, in contrast to *Block generator* that only considers one input-output example at a time. Suppose *Candidate generator* hypothesizes that the solution is a match expression with a guessed scrutinee x. Then, it generates the following hypothesis, distributing the input-output examples of $\square_{in}$ into the two different branches.

$$P_0 = \texttt{rec mul (x : nat, y : nat) : nat = match x with Z -> } \square_1 \texttt{ | S \_ -> } \square_2$$

where $\square_1 = \{(0,1) \mapsto 0\}$ and $\square_2 = \{(1,2) \mapsto 2, (2,1) \mapsto 2\}$. Suppose *Candidate generator* fills the hole $\square_1$ with component x that satisfies the example and moves on to the hole $\square_2$, trying to generate a hypothesis of the form add $(\cdots)$ in that position. Similarly to what *Block generator* did, *Candidate generator* uses the inverse map of add to generate new hypotheses. Because two output examples in $\square_2$ are 2 and there are three inputs of add that lead to the desired output 2 ($\texttt{add}^{-1}(2) = \{(0,2), (1,1), (2,0)\}$), it deduces $9 (= 3^2)$ new hypotheses. Among them, let us consider the following hypothesis

$$P_1 = \texttt{rec mul (x : nat, y : nat) : nat = match x with Z -> x | S \_ -> add } \square_3$$

where $\square_3 = \{(1,2) \mapsto (0,2), (2,1) \mapsto (1,1)\}$. Observing the desired outputs are tuples of length 2, *Candidate generator* distributes the input-output examples into two new holes, generating the following hypothesis.

$$P_2 = \texttt{rec mul (x : nat, y : nat) : nat = match x with Z -> x | S \_ -> add } (\square_4, \square_5)$$

where $\square_4 = \{(1,2) \mapsto 0, (2,1) \mapsto 1\}$ and $\square_5 = \{(1,2) \mapsto 2, (2,1) \mapsto 1\}$. Suppose now it refines the hole $\square_4$ by generating a hypothesis of the form mul $(\cdots)$.

$$P_3 = \texttt{rec mul (x : nat, y : nat) : nat = match x with Z -> x | S \_ -> add ((mul } \square_6), \square_5)$$

Because mul is the target function yet to be defined, we cannot deduce examples for $\square_6$. In such a case, we try enumerating all the components in **C** that can be used as arguments. Recall that we only consider structurally decreasing arguments for recursive calls. For example, mul $(\texttt{S}^{-1}(\texttt{x}), \texttt{y})$ is a valid recursive call as the first parameter decreases. By plugging it into the hole, we obtain

$$\begin{aligned} P_4 = {} & \texttt{rec mul (x : nat, y : nat) : nat =} \\ & \quad \texttt{match x with Z -> x | S \_ -> add (mul } (\texttt{S}^{-1}(\texttt{x}), \texttt{y}), \square_5) \end{aligned}$$

Whenever a hypothesis containing recursive calls is generated, *Candidate generator* checks the feasibility of the hypothesis. It first performs symbolic evaluation interleaved with concrete evaluation with each input example on the hypothesis to obtain blocks. Our symbolic evaluation obeys the following rules.

- The body of the hypothesis is substituted into every position of a recursive call, and actual parameters are substituted for formal parameters.
- Every scrutinee in a match expression is concretely evaluated with a given input to take a branch.
- Calls to external operators and holes are left unchanged.

We call this process *unfolding*. As a result of unfolding, we obtain an open block, i.e., a block possibly with holes. Let us denote $\rightarrow^*$ as one or more steps of the symbolic evaluation. The followings show how to derive a block B_j from the hypothesis for each j-th input-output example associated with $\square_{in}$.

$(0,1) \mapsto 0$: mul (x,y)
$\rightarrow$ match x with Z -> x | S _ -> add (mul (S^{-1}(x), y), $\square_5$)
$\rightarrow$ match 0 with Z -> x | S _ -> add (mul (S^{-1}(x), y), $\square_5$)
$\rightarrow$ x (= B_0)
$(1,2) \mapsto 2$: mul (x,y)
$\rightarrow$ match x with Z -> x | S _ -> add (mul (S^{-1}(x), y), $\square_5$)
$\rightarrow^*$ add (mul (S^{-1}(x), y), $\square_5$)
$\rightarrow$ add (match S^{-1}(x) with Z -> S^{-1}(x) | S _ -> add (mul (S^{-2}(x), y), $\square_5$), $\square_5$)
$\rightarrow$ add (match 0 with Z -> S^{-1}(x) | S _ -> add (mul (S^{-2}(x), y), $\square_5$), $\square_5$)
$\rightarrow$ add (S^{-1}(x), $\square_5$) (= B_1)
$(2,1) \mapsto 2$: mul (x,y)
$\rightarrow$ match x with Z -> x | S _ -> add (mul (S^{-1}(x), y), $\square_5$)
$\rightarrow^*$ add (mul (S^{-1}(x), y), $\square_5$)
$\rightarrow$ add (match S^{-1}(x) with Z -> S^{-1}(x) | S _ -> add (mul (S^{-2}(x), y), $\square_5$), $\square_5$)
$\rightarrow$ add (match 1 with Z -> S^{-1}(x) | S _ -> add (mul (S^{-2}(x), y), $\square_5$), $\square_5$)
$\rightarrow^*$ add (add (mul (S^{-2}(x), y), $\square_5$), $\square_5$)
$\rightarrow$ add (add (match S^{-2}(x) with Z -> S^{-2}(x) | S _ -> add (mul (S^{-3}(x), y), $\square_5$), $\square_5$), $\square_5$)
$\rightarrow^*$ add (add (S^{-2}(x), $\square_5$), $\square_5$) (= B_2)

where S^{-2}(x) is a shorthand for $S^{-1}(S^{-1}$(x)). Then, for all j, it checks if each block B_j can be identical to another block in $\mathbf{B}_j$ by properly substituting each hole. B_0, which is x, is identical to x in $\mathbf{B}_0$. B_1, which is add (S^{-1}(x), $\square_5$), can be identical to add (S^{-1}(x), y) in $\mathbf{B_1}$. Lastly, B_2, which is add (add (S^{-2}(x), $\square_5$), $\square_5$), can be identical to add (add (S^{-2}(x), y), y) in $\mathbf{B}_2$. This matching process can be efficiently done by traversing the version spaces of the blocks. The fact that the hypothesis can be *unfolded* into blocks satisfying the examples suggests that we may find a solution if we further refine the hypothesis. Thus, P_4 is determined to be feasible. Next, the hole $\square_5$ can be filled with y, which is a component satisfying the example over the hole, and we find the solution.

Feedback Loop for Guaranteeing Search Completeness. Although the block-based pruning presented may be *unsound* in some cases, the overall algorithm eventually finds a solution if it exists. A feasible hypothesis may be mistakenly rejected if *Block generator* misses some satisfying blocks because of its limited search in a finitized space. Trio uses a feedback loop to avoid such unsound pruning. If a solution cannot be found using a current set of components, Trio will add larger components into the component pool and repeat the entire process, so that *Block generator* can generate more blocks and hopefully avoid mistakenly rejecting correct hypotheses.

Also, when constructing inverse maps, despite restricting the domain of external functions to be the set of values each of which is not greater than the greatest value in the examples, we do not miss a solution involving external functions. This is because the bottom-up enumerator will eventually enumerate necessary function call expressions of finite size.

3 PROBLEM DEFINITION

In this section, we define our problem of inductive synthesis of recursive functional programs. We first define an ML-like functional language in which we synthesize programs.

3.1 Language

We consider an idealized functional language similar to the core of ML. Our target language features algebraic data types and recursive functions with the syntax definition depicted in Fig. 2. Programs P are recursive functions whose bodies are expressions e. Application is written $e_1\ e_2$, κ ranges

$$\begin{array}{llll} e \in \mathit{Exp} & \text{(Expressions)} & \kappa \in \mathit{Constructors} & \text{(Constructors)} \\ v \in \mathit{Val} & \text{(Values)} & \square_u \in \mathit{Val} \xrightarrow{\text{fin}} \mathit{Val} & \text{(Input-Output Examples)} \end{array}$$

$$\begin{array}{lcl} P & ::= & \mathsf{rec}\ \mathsf{f}(\mathsf{x}) = e \\ e & ::= & x \mid e_1\ e_2 \mid \kappa(e_1, \cdots, e_{a(\kappa)}) \mid \kappa^{-1}(e) \mid (e_1, \cdots, e_n) \mid e.n \mid \mathsf{rec}\ f(x) = e \\ & \mid & \mathsf{match}\ e\ \mathsf{with}\ \overline{\kappa_j\ \_ \longrightarrow e_j}^{k} \mid \square_u \\ v & ::= & \kappa(v_1, \cdots, v_k) \mid (v_1, \cdots, v_m) \mid \mathsf{rec}\ f(x) = e \\ \sigma & ::= & \{\} \mid \{x \mapsto v\} \cup \sigma \end{array}$$

Fig. 2. Our ML-like language

over data type constructors, $a(\kappa)$ denotes the arity of κ, κ^{-1} denotes a destructor which extracts all the subcomponents of a constructor application of κ as a tuple. An expression $e.n$ projects the n-th component of a tuple. We use ML-style pattern match expressions. We use $\overline{\kappa_j\ \_ \longrightarrow e_j}^{k}$ to denote $\kappa_1\ \_ \longrightarrow e_1 \mid \cdots \mid \kappa_k\ \_ \longrightarrow e_k$. A hole is written $\square_u$, where u is the hole name, which we tacitly assume is unique. Each hole is associated with input-output examples, a finite function from input values to output values. Values v are made up of constructor values for data types, tuples, and recursive functions. Recursive functions can be used as input examples when synthesizing higher-order functions but cannot be used as output examples. Environments σ map variables to values. For conciseness, we assume that all functions take a single argument, which does not harm the expressivity of the language since we can represent multiple inputs as a single tuple.

3.2 Notations

We will use some notations throughout the remaining sections. An *open hypothesis* (resp. open expression) is a program (resp. expression) that contains one or more holes. A *closed hypothesis* is a program that does not contain any holes. We will use the fixed variables f and x to denote the target function and its formal parameter, respectively. We use $\sigma \vdash P \Rightarrow v$ to denote the standard multi-step call-by-value operational semantics of a program P without holes under environment σ.

3.3 Problem Definition

Given an environment σ that provides definitions of *external* functions and an initial open hypothesis $P_{in} = \mathsf{rec}\ \mathsf{f}(\mathsf{x}) = \square_{in}$ where $\square_{in} = \bigcup_{1 \le j \le n} \{i_j \mapsto o_j\}$ represents input-output examples that should be satisfied by the function body, our goal is to find a closed hypothesis of form $P = \mathsf{rec}\ \mathsf{f}(\mathsf{x}) = e$ that satisfies the input-output examples of $\square_{in}$. Formally, $\forall 1 \le j \le n.\ \sigma[\mathsf{f} \mapsto \mathsf{rec}\ \mathsf{f}(\mathsf{x}) = e, \mathsf{x} \mapsto i_j] \vdash P \Rightarrow o_j$ (denoted $P \models_\sigma \square_{in}$). We just use the user-provided external functions without inventing new ones.

4 ALGORITHM

This section formally describes our algorithm, inspired by the previous methods by Lee [2021] and Feser et al. [2015].

4.1 Overall Algorithm

Fig. 1 shows the high-level structure of our algorithm. The algorithm takes as input an environment σ that provides definitions of external functions, which we tacitly assume to be globally accessible throughout the algorithm, an initial open hypothesis with input-output examples, and initial component size n. Finally, it returns a program P that satisfies the input-output examples. With

Algorithm 1 The Trio Algorithm

```
Require: (Global variable) environment $\sigma$ that provides definitions of external functions
Require: Initial component size $n$
Require: A hypothesis $P_{in}$ = rec f(x) = $\Box_{in}$ where $\Box_{in} = \bigcup_{1 \leq j \leq n} \{i_j \mapsto o_j\}$
Ensure: A solution program $P$
 1: C := $\emptyset$
 2: repeat
 3:     C := ComponentGeneration(C, $\Box_{in}$, $n$)        } Bottom-Up      ▷ C : $\mathcal{P}(Exp)$
 4:     $\mathcal{I}$ := LibrarySampling(C, $\Box_{in}$)              } Enumerator     ▷ $\mathcal{I} : \mathcal{P}(Val \times Val \times Val)$
 5:     C_simple := C − RecursiveOrMatchExprs(C)     } Block Generator
 6:     B := BlockGen(C_simple, $\mathcal{I}$, $\Box_{in}$)            }                ▷ B : $Val \times Val \to \mathcal{P}(Exp)$
 7:     $Q$ := {$P_{in}$}                                  } Candidate Generator (lines 7–23)
 8:     while $Q \neq \emptyset$ do
 9:         remove $P$ such that $P$ has minimal cost from $Q$      ▷ Using a cost model in Section 5.2
10:         if Holes($P$) = $\emptyset$ then
11:             if $P \models_\sigma \Box_{in}$ then return $P$
12:             else continue
13:             end if
14:         end if
15:         Pick a hole $\Box_u$ in $P$
16:         for $e \in$ Deduce(C, $\mathcal{I}$, $\Box_u$) do
17:             $P'$ := $P[e/\Box_u]$
18:             if Holes($P'$) = $\emptyset$ then insert $P'$ into $Q$
19:             else if BlockConsistent($P'$, B, $\Box_{in}$) then
20:                 insert $P'$ into $Q$
21:             end if
22:         end for
23:     end while
24:     $n$ := $n$ + 1
25: until false
```

initially empty component set **C**, the main loop of our synthesis procedure (line 2 – 25) is repeated until a solution is found. The loop starts by invoking the ComponentGeneration procedure (line 3) which takes a current component pool **C**, the input-output examples $\Box_{in}$, and the target component size n. The procedure generates new components by composing existing components in **C**. It applies the standard pruning technique based on observational equivalence with respect to the input examples. Expressions with recursive calls to the target function being synthesized can be included in the resulting component pool. Because we cannot evaluate such recursive components as the function is unknown yet, we cannot apply the observational equivalence reduction based on their outputs. Instead, we exploit *functional congruence*, i.e., the same input to the function always results in the same output. For example, we do not maintain both of f 2 and f (1 + 1) in the component pool. Next, the LibrarySampling procedure (Section 4.2) is invoked to derive an inverse map for each function expression in the component pool (line 4). Next, the BlockGen procedure is invoked to obtain satisfying blocks for each input-output example (line 6). Each block must be recursion- and conditional-free because the target function is unknown yet and conditionals are not necessary when it comes to a single input-output example. Therefore, any components containing recursive calls and conditionals must not be used in blocks. We exclude such components from **C** (line 5), and provide the reduced component set to the BlockGen procedure (Section 4.4). With inverse maps $\mathcal{I}$ and blocks **B**, the inner loop (lines 8 – 23) iteratively processes elements in the priority queue Q. The priority queue Q contains hypotheses (initially only P_{in}), and is sorted according to the cost (Section 5.2) of each hypothesis. In each iteration, we pick a minimum-cost hypothesis P from the queue. If P is closed (line 10) and correct with respect to the top-level input-output examples, P is returned as a solution (line 11). Otherwise, we continue investigating other hypotheses in the queue. If a chosen hypothesis P is open, we pick a hole $\Box_u$ in P (line 15). Then, the Deduce procedure (Section 4.3) returns possible replacements for the hole $\Box_u$ (line 16). A replacement e for the hole may be a closed expression satisfying the example of $\Box_u$, or an open expression with new holes. For each replacement e, we obtain a new hypothesis P' by replacing the hole with e (line 17). If P' is closed, there are no unknowns left to be synthesized (line 18). Hence, we add P' into the queue, so that its correctness can be checked in the next iterations. If P' is open, we check its consistency with the blocks $\mathbf{B}$ before adding it to the queue (line 19) by invoking the BlockConsistent procedure (Section 4.5). If the queue becomes empty before a solution is found, we increase the component size n by 1 (line 24) and restart the main loop.

Our algorithm is sound and complete in that it finds a program correct with respect to the given input-output examples if it exists in the search space.

THEOREM 4.1. *Algo. 1 finds a solution to a given synthesis problem if it exists.*

PROOF. Available in the Supplemental Material at the ACM DL. □

4.2 Getting Inverse Maps of External Functions by Library Sampling

This section describes the LibrarySampling procedure that derives a set $\mathcal{I}$ whose each element is a triple (g, v_o, v_i) where g is a function value and v_i and v_o are non-function values, meaning that $\sigma \vdash g\ v_i \Rightarrow v_o$. We will write (g, v_o, v_i) as $g^{-1}(v_o) = v_i$.

Given the component pool $\mathbf{C}$ and the top-level input-output examples $\Box_{in} = \bigcup_{1 \le j \le n} \{i_j \mapsto o_j\}$ as input, we first compute a finite domain $D = \{v \in \mathit{Val} \mid \exists 1 \le j \le n.\ v \sqsubseteq i_j\}$ where $\sqsubseteq$ denotes a well-founded ordering on values. In our implementation, we represent values as abstract syntax trees and use the subtree relation. Using the values in D as inputs, we compute a set of inverse maps as follows:

$$\mathcal{I} = \{g^{-1}(v_o) = v_i \mid g, v_o \in \mathit{Val}, v_i \in D, \exists e \in \mathbf{C}, 1 \le j \le n.\ \sigma[\mathtt{x} \mapsto i_j] \vdash e \Rightarrow g, \sigma \vdash g\ v_i \Rightarrow v_o\}$$

The use of the component pool $\mathbf{C}$ is for computing inverse maps of functions provided as input examples, which is useful for synthesizing higher-order functions.

Example 4.2. Consider the following hypothesis.

```
type nat = Z | S of nat
rec f (x : (nat -> nat) * nat) : nat = □in
```

where $\Box_{in} = \{(\text{rec one (n) = S(Z)}, 1) \mapsto 1, (\text{rec inc (n) = S(n)}, 0) \mapsto 1\}$. The solution is

```
rec f (x : (nat -> nat) * nat) : nat = x.1 x.2
```

Suppose the component pool $\mathbf{C} = \{\mathtt{x.1}, \mathtt{x.2}\}$. The domain D is $\{0, 1\}$. We derive $(\text{rec one(n) = S(Z)})^{-1}(1) = 0 \in \mathcal{I}$ because $\sigma[\mathtt{x} \mapsto (\text{rec one(n) = S(Z)}, 1)] \vdash \mathtt{x.1} \Rightarrow \text{rec one(n) = S(Z)}$ and $\sigma \vdash (\text{rec one(n) = S(Z)})\ 0 \Rightarrow 1$. In a similar manner, we conclude $(\text{rec inc(n) = S(n)})^{-1}(1) = 0 \in \mathcal{I}$. In conclusion,

$$\mathcal{I} = \{(\text{rec one(n) = S(Z)})^{-1}(1) = 0, (\text{rec one(n) = S(Z)})^{-1}(1) = 1, (\text{rec inc(n) = S(n)})^{-1}(1) = 0\}.$$

We consider D to be the domain for the following reason. We permit recursive calls on values that are strictly smaller than the input to ensure that our synthesized programs terminate similarly to prior work [Miltner et al. 2022]. In other words, if v_{in} is provided as an input, recursive calls to f

$$\frac{}{\{e \in \mathbf{C} \mid e \models_\sigma \square_u\} \subseteq \text{Deduce}(\mathbf{C}, \mathcal{I}, \square_u)}\ \text{D\_COMPONENT}$$

$$\frac{}{\{e \in \mathbf{C} \mid \text{e contains recursive calls.}\} \subseteq \text{Deduce}(\mathbf{C}, \mathcal{I}, \square_u)}\ \text{D\_REC}$$

$$\frac{\square_u = \bigcup_{1 \le j \le n}\{i_j \mapsto \kappa(v_{1j}, \cdots, v_{kj})\} \qquad \forall 1 \le m \le k.\ \square_{u_m} = \bigcup_{1 \le j \le n}\{i_j \mapsto v_{mj}\}}{\kappa(\square_{u_1}, \cdots, \square_{u_k}) \in \text{Deduce}(\mathbf{C}, \mathcal{I}, \square_u)}\ \text{D\_CTOR}$$

$$\frac{\square_u = \bigcup_{1 \le j \le n}\{i_j \mapsto o_j\} \qquad \kappa \in \mathit{Constructors} \qquad \square_{u'} = \bigcup_{1 \le j \le n}\{i_j \mapsto \kappa(o_j)\}}{\kappa^{-1}(\square_{u'}) \in \text{Deduce}(\mathbf{C}, \mathcal{I}, \square_u)}\ \text{D\_DTOR}$$

$$\frac{\square_u = \bigcup_{1 \le j \le n}\{i_j \mapsto o_j\} \qquad e \in \mathbf{C} \qquad m \in \mathbb{N} \qquad \forall 1 \le j \le n.\ \sigma[\mathsf{x} \mapsto i_j] \vdash e.m \Rightarrow o_j}{e.m \in \text{Deduce}(\mathbf{C}, \mathcal{I}, \square_u)}\ \text{D\_PROJ}$$

$$\frac{\square_u = \bigcup_{1 \le j \le n}\{i_j \mapsto (v_{1j}, \cdots, v_{kj})\} \qquad \forall 1 \le m \le k.\ \square_{u_m} = \bigcup_{1 \le j \le n}\{i_j \mapsto v_{mj}\}}{(\square_{u_1}, \cdots, \square_{u_k}) \in \text{Deduce}(\mathbf{C}, \mathcal{I}, \square_u)}\ \text{D\_TUPLE}$$

$$\frac{\begin{array}{c}\square_u = \bigcup_{1 \le j \le n}\{i_j \mapsto o_j\} \qquad e \in \mathbf{C} \\ \forall 1 \le m \le k.\ \square_{u_m} = \bigcup_{j \in I_m}\{i_j \mapsto o_j\}\ \text{ where }\ I_m = \{j \mid 1 \le j \le n, \sigma[\mathsf{x} \mapsto i_j] \vdash e \Rightarrow \kappa_m(\_)\}\end{array}}{\text{match } e \text{ with } \overline{\kappa_i\ \_ \to \square_{u_i}}^{k} \in \text{Deduce}(\mathbf{C}, \mathcal{I}, \square_u)}\ \text{D\_MATCH}$$

$$\frac{\begin{array}{c}\square_u = \bigcup_{1 \le j \le n}\{i_j \mapsto o_j\} \quad \square_{u_1} = \bigcup_{1 \le j \le n}\{i_j \mapsto g_j\} \\ \square_{u_2} = \bigcup_{1 \le j \le n}\{i_j \mapsto v_j\} \quad \forall 1 \le j \le n.\ {g_j}^{-1}(o_j) = v_j \in \mathcal{I}\end{array}}{\square_{u_1}\ \square_{u_2} \in \text{Deduce}(\mathbf{C}, \mathcal{I}, \square_u)}\ \text{D\_EXTCALL}$$

Fig. 3. Inference rules for Deduce

can only be applied to values v when $v \sqsubset v_{in}$. Inputs to a function often flow to other functions called inside of it. Therefore, it is likely that $\mathcal{I}$ captures input-output behaviors of library functions that can be observed during the evaluation of the desired program with the user-provided input examples.

4.3 The Deduce Procedure

We now describe the Deduce procedure that returns a set of expressions that are either closed or open as possible replacements for a given hole $\square_u$.

Deduce$(\mathbf{C}, \mathcal{I}, \square_u)$ is the smallest set of expressions satisfying the constraints depicted in Fig. 3. The D_COMPONENT rule says if we have components in $\mathbf{C}$ that immediately satisfy $\square_u$, every such component can fill the hole ($e \models_\sigma \square_u$ denotes $\forall i \mapsto o \in \square_u.\ \sigma[\mathsf{x} \mapsto i] \vdash e \Rightarrow o$). D_REC indicates that all recursive components in $\mathbf{C}$ are considered potential replacements for the hole. This rule is under an optimistic assumption that any recursive expressions whose semantics is unknown yet may satisfy the hole, although in reality not all recursive expressions can do. Later, any hypothesis containing a recursive call that is determined to be infeasible will be discarded (will be detailed in Section 4.5). D_CTOR generates new examples for arguments of a constructor application. If the example values in the hole $\square_u$ consist of constructor values with a shared constructor κ of arity k, then it creates k new examples constraints over the k arguments of the constructor value. D_DTOR creates a new example for the argument of a destructor value. The new example consists of constructor values with a shared constructor κ where κ can be any constructor. D_PROJ creates closed expressions as replacements for the hole. D_TUPLE is for creating new examples corresponding to arguments that must be synthesized for a tuple expression. The deductive reasoning process is similar to that of D_CTOR. D_MATCH first identifies components that can be used as scrutinees. Then, for each match expression whose scrutinee is such a component, it distributes the given examples in the hole to each branch. Lastly, D_EXTCALL uses the inverse maps of external functions. For example, for an input-output example $i \mapsto o$, if we can find a triple $g^{-1}(o) = v$ in $\mathcal{I}$, then we can deduce an example $i \mapsto g$ for the function part and another example $i \mapsto v$ for the argument part.

Example 4.3. Consider the overview example in Section 2. Let us denote the target function as f. Suppose we have a hole $\square_u = \{(1,2) \mapsto 2\}$ and a component pool $\mathbf{C} = \{\ \mathsf{x}, \mathsf{f}\ (\mathsf{S}^{-1}(\mathsf{x}.1), \mathsf{x}.2), 2, \mathsf{add}\ \}$. We can deduce the following constraints by applying the rules in Fig. 3.

By D_COMPONENT, $2 \in \text{Deduce}(\mathbf{C}, \mathcal{I}, \square_u)$ $\quad (\because 2 \in \mathbf{C}, 2 \models_\sigma \square_u)$

By D_REC, $\mathsf{f}\ (\mathsf{S}^{-1}(\mathsf{x}.1), \mathsf{x}.2) \in \text{Deduce}(\mathbf{C}, \mathcal{I}, \square_u)$

By D_CTOR, $\mathsf{S}(\square_{u_1}) \in \text{Deduce}(\mathbf{C}, \mathcal{I}, \square_u)$ $\quad (\square_{u_1} = \{(1,2) \mapsto 1\})$

By D_DTOR, $\mathsf{S}^{-1}(\square_{u_2}) \in \text{Deduce}(\mathbf{C}, \mathcal{I}, \square_u)$ $\quad (\square_{u_2} = \{(1,2) \mapsto 3\})$

By D_PROJ, $\mathsf{x}.2 \in \text{Deduce}(\mathbf{C}, \mathcal{I}, \square_u)$ $\quad (\because \mathsf{x} \in \mathbf{C}, \mathsf{x}.2 \models_\sigma \square_u)$

By D_MATCH, $\mathsf{match}\ 2\ \mathsf{with}\ \mathsf{Z} \to \square_{u_3} \mid \mathsf{S}\ \_ \to \square_u \in \text{Deduce}(\mathbf{C}, \mathcal{I}, \square_u)$ $\quad (\square_{u_3} = \emptyset)$

$(\because 2 \in \mathbf{C}, \sigma[\mathsf{x} \mapsto (1,2)] \vdash 2 \Rightarrow \mathsf{S}\ \_)$

By D_EXTCALL,

$\square_{u_4}\ \square_{u_5} \in \text{Deduce}(\mathbf{C}, \mathcal{I}, \square_u)$ $\quad (\square_{u_4} = \{(1,2) \mapsto \mathsf{rec\ add}\ \ldots\}, \square_{u_5} = \{(1,2) \mapsto (2,0)\})$

$\square_{u_4}\ \square_{u_6} \in \text{Deduce}(\mathbf{C}, \mathcal{I}, \square_u)$ $\quad (\square_{u_6} = \{(1,2) \mapsto (1,1)\})$

$\square_{u_4}\ \square_{u_7} \in \text{Deduce}(\mathbf{C}, \mathcal{I}, \square_u)$ $\quad (\square_{u_7} = \{(1,2) \mapsto (0,2)\})$

$(\because \{(\mathsf{rec\ add} \cdots)^{-1}(2) = (2,0), (\mathsf{rec\ add} \cdots)^{-1}(2) = (1,1), (\mathsf{rec\ add} \cdots)^{-1}(2) = (0,2)\} \subseteq \mathcal{I})$

Note that we cannot apply the D_TUPLE rule because $\square_u$ does not contain any tuple. Also, when applying the D_MATCH rule, we cannot use the components x and $\mathsf{f}\ (\mathsf{S}^{-1}(\mathsf{x}.1), \mathsf{x}.2)$ as a scrutinee because neither of them evaluates to a constructor application (in particular, $\mathsf{f}\ (\mathsf{S}^{-1}(\mathsf{x}.1), \mathsf{x}.2)$ cannot evaluate to a concrete value as f is not defined yet). The following is the smallest solution satisfying the constraints over $\text{Deduce}(\mathbf{C}, \square_u)$.

$$\begin{aligned}\text{Deduce}(\mathbf{C}, \mathcal{I}, \square_u) = \{&2, \mathsf{f}\ (\mathsf{S}^{-1}(\mathsf{x}.1), \mathsf{x}.2), \mathsf{S}(\square_{u_1}), \mathsf{S}^{-1}(\square_{u_2}), \mathsf{x}.2, \mathsf{match}\ 2\ \mathsf{with}\ \mathsf{Z} \to \square_{u_3} \mid \mathsf{S}\_ \to \square_u,\\ &\square_{u_4}\ \square_{u_5}, \square_{u_4}\ \square_{u_6}, \square_{u_4}\ \square_{u_7}\}\end{aligned}$$

The Deduce procedure is *sound* in the following sense.

Definition 4.4 (Soundness of Deduction). Let $\square_u$ be a set of input-output examples, and let $\mathbf{C}$ and $\mathcal{I}$ be a set of components and a set of inverse maps, respectively. If there exists an expression satisfying $\square_u$, for every open expression $e \in \text{Deduce}(\mathbf{C}, \mathcal{I}, \square_u)$, for every hole $\square_{u'}$ in e, there exists an expression satisfying the hole $\square_{u'}$.

Intuitively, the deduction procedure is sound if there exists a solution to a synthesis task, then there also exists a solution to every synthesis subtask derived from the original synthesis task.

THEOREM 4.5. *Without using the D_EXTCALL rule, the Deduce procedure is sound.*

PROOF. Available in the Supplemental Material at the ACM DL. □

The following example shows that the Deduce procedure can be unsound when using the D_EXTCALL rule.

$$\frac{\Box_u = \bigcup_{1\le j\le n}\{i_j \mapsto o_j\}}{\mathsf{BlockGen}(\mathrm{C}, \mathcal{I}, \Box_u) = \{(i_j, o_j) \mapsto \mathsf{Blocks}(\mathrm{C}, \mathcal{I}, i_j \mapsto o_j) \mid 1 \le j \le n\}}\ \text{B\_GEN}$$

$$\frac{\mathsf{SimpleBlocks} = \{e \in \mathrm{C} \mid \sigma[\mathsf{x} \mapsto i] \vdash e \Rightarrow o\}}{\mathsf{Blocks}(\mathrm{C}, \mathcal{I}, i \mapsto o) = \bigcup\ \mathsf{CompoundBlocks}(\mathrm{C}, \mathcal{I}, i \mapsto o) \cup \mathsf{SimpleBlocks}}\ \text{B\_GEN\_PER\_EX}$$

$$\frac{\kappa(\Box_{u_1}, \cdots, \Box_{u_k}) \in \mathsf{Deduce}(\mathrm{C}, \mathcal{I}, \{i \mapsto o\}) \quad \forall 1 \le m \le k.\ \widetilde{e_m} = \mathsf{Blocks}(\mathrm{C}, \mathcal{I}, \Box_{u_m})}{\kappa(\widetilde{e_1}, \cdots, \widetilde{e_k}) \subseteq \mathsf{CompoundBlocks}(\mathrm{C}, \mathcal{I}, i \mapsto o)}\ \text{B\_CTOR}$$

$$\frac{\kappa^{-1}(\Box_u) \in \mathsf{Deduce}(\mathrm{C}, \mathcal{I}, \{i \mapsto o\}) \quad \widetilde{e} = \mathsf{Blocks}(\mathrm{C}, \mathcal{I}, \Box_u)}{\kappa^{-1}(\ \widetilde{e}\ ) \subseteq \mathsf{CompoundBlocks}(\mathrm{C}, \mathcal{I}, i \mapsto o)}\ \text{B\_DTOR} \qquad \frac{e.n \in \mathsf{Deduce}(\mathrm{C}, \mathcal{I}, \{i \mapsto o\})}{e.n \in \mathsf{CompoundBlocks}(\mathrm{C}, \mathcal{I}, i \mapsto o)}\ \text{B\_PROJ}$$

$$\frac{(\Box_{u_1}, \cdots, \Box_{u_k}) \in \mathsf{Deduce}(\mathrm{C}, \mathcal{I}, \{i \mapsto o\}) \qquad \forall 1 \le m \le k.\ \widetilde{e_m} = \mathsf{Blocks}(\mathrm{C}, \mathcal{I}, \Box_{u_m})}{(\widetilde{e_1}, \cdots, \widetilde{e_k}) \subseteq \mathsf{CompoundBlocks}(\mathrm{C}, \mathcal{I}, i \mapsto o)}\ \text{B\_TUPLE}$$

$$\frac{\Box_{u_1}\ \Box_{u_2} \in \mathsf{Deduce}(\mathrm{C}, \mathcal{I}, \{i \mapsto o\}) \qquad \forall 1 \le m \le 2.\ \tilde{E}_m = \mathsf{Blocks}(\mathrm{C}, \mathcal{I}, \Box_{u_m})}{(\widetilde{e_1}\ \widetilde{e_2}) \subseteq \mathsf{CompoundBlocks}(\mathrm{C}, \mathcal{I}, i \mapsto o)}\ \text{B\_APP}$$

Fig. 4. Inference rules for BlockGen

Example 4.6. Recall Example 4.2. Let us denote the first and second input examples in $\Box_{in}$ as i_1 and i_2 respectively (i.e., $i_1 = (\mathsf{rec\ one\ (n) = S(Z)}, 1)$, $i_2 = (\mathsf{rec\ inc\ (n) = S(n)}, 0)$). We can deduce the following fact by applying the D_EXTCALL rule because $(\mathsf{rec\ one} \cdots)^{-1}(1) = 0 \in \mathcal{I}$.

$$\Box_{u_1}\ \Box_{u_2} \in \mathsf{Deduce}(\mathrm{C}, \Box_{in}) \qquad\qquad (\Box_{u_1} = \{i_1 \mapsto \mathsf{rec\ one} \cdots, i_2 \mapsto \mathsf{rec\ one} \cdots\}, \Box_{u_2} = \{i_1 \mapsto 0, i_2 \mapsto 0\})$$

We cannot synthesize an expression satisfying the hole $\Box_{u_1}$. That is because we cannot synthesize an expression that evalutes to the one function under the environment where x is bound to i_2 (the only available function is inc). Recall that we do not synthesize any new auxiliary functions.

4.4 Constructing Blocks From Each Input-Output Example

This section describes the BlockGen procedure for computing satisfying blocks for each input-output example in $\Box_{in}$. The set of blocks is stored in a *version space* [Gulwani 2011] which is a compact representation of expressions.

We begin with the definition of version spaces.

Definition 4.7 (Version Space). A version space is either

- A union: $\bigcup \mathbf{V}$ where $\mathbf{V}$ is a set of version spaces
- An expression
- An application: written $(\widetilde{e_1}\ \widetilde{e_2})$ where $\widetilde{e_i}$ are version spaces
- A tuple: written $(\widetilde{e_1}, \cdots, \widetilde{e_k})$ where $\widetilde{e_i}$ are version spaces
- A constructor: written $\kappa(\widetilde{e_1}, \cdots, \widetilde{e_k})$ where $\widetilde{e_i}$ are version spaces and κ is a constructor
- A destructor: written $\kappa^{-1}(\widetilde{e})$ where $\widetilde{e}$ is a version space and κ is a constructor
- The empty set, $\emptyset$

A version space can be understood as an E-graph where each node represents a set of expressions. Each leaf node represents a single expression, and they are composed into larger sets. The union operator $\bigcup$ symbolizes a nondeterministic choice between multiple expressions, allowing version spaces to compactly represent huge sets of expressions.

Example 4.8. The version space $(\mathsf{add}\ (\bigcup\{\mathsf{x}, \mathsf{Z}\}, \bigcup\{\mathsf{x}, \mathsf{Z}\})$ encodes four different expressions: $\mathsf{add\ (x, x)}$, $\mathsf{add\ (x, Z)}$, $\mathsf{add\ (Z, x)}$, and $\mathsf{add\ (Z, Z)}$.

The set of expressions encoded by a version space is defined as follows:

Definition 4.9. The set represented by a version space is written $\llbracket\, \widetilde{e}\, \rrbracket$ and is defined recursively as:

$$\llbracket e \rrbracket = e \quad (e \text{ is an expression}) \qquad \llbracket \emptyset \rrbracket = \emptyset \qquad \llbracket \bigcup \mathbf{V} \rrbracket = \{ e \in \llbracket \widetilde{e} \rrbracket \mid \widetilde{e} \in \mathbf{V} \}$$

$$\llbracket (\widetilde{e_1}, \cdots, \widetilde{e_k}) \rrbracket = \{ (e_1, \cdots, e_k) \mid \forall 1 \le j \le k.\ e_j \in \llbracket \widetilde{e_j} \rrbracket \}$$

$$\llbracket \kappa(\widetilde{e_1}, \cdots, \widetilde{e_k}) \rrbracket = \{ \kappa(e_1, \cdots, e_k) \mid \forall 1 \le j \le k.\ e_j \in \llbracket \widetilde{e_j} \rrbracket \}$$

$$\llbracket \kappa^{-1}(\widetilde{e}) \rrbracket = \{ \kappa^{-1}(e) \mid e \in \llbracket \widetilde{e} \rrbracket \} \qquad \llbracket (\widetilde{e_1}\ \widetilde{e_2}) \rrbracket = \{ (e_1\ e_2) \mid \forall 1 \le j \le 2.\ e_j \in \llbracket \widetilde{e_j} \rrbracket \}$$

With this in mind, we are ready to describe how to obtain a version space of blocks. Given a set C of components and the top-level input-output examples $\Box_{in}$, the BlockGen procedure computes the smallest version spaces satisfying the constraints in Fig. 4. The result maps each input-output example to a version space of satisfying blocks. In the b_gen_per_ex rule, Blocks(C, $i \mapsto o$) denotes the version space of blocks satisfying a single input-output example $i \mapsto o$. The other rules depict how to compute a version space for a single example. SimpleBlocks denotes a set of component expressions satisfying that example. CompoundBlocks denotes a set of version spaces each of which is not a single expression. We reuse the Deduce procedure to obtain CompoundBlocks, which can be derived by the other remaining rules b_ctor, $\cdots$, b_app. Note that the given set of components C only includes recursion- and conditional-free expressions (by line 5 in Algo. 1), and there are no rules for deriving version spaces containing match expressions and recursive calls, thereby ensuring that the resulting version space represents a set of blocks.

Example 4.10. Recall Deduce(C, $\mathcal{I}$, $\Box_u$) in Example 4.3. We describe how to obtain a version space of blocks for the input-output example $\Box_u = \{(1, 2) \mapsto 2\}$ using the rules in Fig. 4. Blocks(C, $\mathcal{I}$, $\Box_u$) is computed as follows: first, by the b_gen_per_ex rule,

$$\mathsf{Blocks}(\mathrm{C}, \mathcal{I}, \Box_u) = \bigcup \mathsf{CompoundBlocks}(\mathrm{C}, \mathcal{I}, \Box_u) \cup \mathsf{SimpleBlocks}$$

where SimpleBlocks = {2} because 2 is the only component satisfying the example. We can deduce the following constraints over CompoundBlocks(C, $\mathcal{I}$, $\Box_u$) as follows:

By b_ctor, S(Blocks(C, $\mathcal{I}$, $\Box_{u_1}$)) ⊆ CompoundBlocks(C, $\mathcal{I}$, $\Box_u$) $\quad$ (∵ S($\Box_{u_1}$) ∈ Deduce(C, $\mathcal{I}$, $\Box_u$))

By b_dtor, S^{-1}(Blocks(C, $\mathcal{I}$, $\Box_{u_2}$)) ⊆ CompoundBlocks(C, $\mathcal{I}$, $\Box_u$) $\quad$ (∵ S^{-1}($\Box_{u_2}$) ∈ Deduce(C, $\mathcal{I}$, $\Box_u$))

By b_proj, x.2 ∈ CompoundBlocks(C, $\mathcal{I}$, $\Box_u$) $\quad$ (∵ x.2 ∈ Deduce(C, $\mathcal{I}$, $\Box_u$))

By b_app

$$(\mathsf{Blocks}(\mathrm{C}, \mathcal{I}, \Box_{u_4})\ \mathsf{Blocks}(\mathrm{C}, \mathcal{I}, \Box_{u_5})) \subseteq \mathsf{CompoundBlocks}(\mathrm{C}, \mathcal{I}, \Box_u)$$
$$(\mathsf{Blocks}(\mathrm{C}, \mathcal{I}, \Box_{u_4})\ \mathsf{Blocks}(\mathrm{C}, \mathcal{I}, \Box_{u_6})) \subseteq \mathsf{CompoundBlocks}(\mathrm{C}, \mathcal{I}, \Box_u)$$
$$(\mathsf{Blocks}(\mathrm{C}, \mathcal{I}, \Box_{u_4})\ \mathsf{Blocks}(\mathrm{C}, \mathcal{I}, \Box_{u_7})) \subseteq \mathsf{CompoundBlocks}(\mathrm{C}, \mathcal{I}, \Box_u)$$

where $\Box_{u_1}, \cdots, \Box_{u_7}$ are the ones defined in Example 4.3. We keep applying the rules to generate constraints over Blocks(C, $\mathcal{I}$, $\Box_{u_1}$), $\cdots$, Blocks(C, $\mathcal{I}$, $\Box_{u_7}$). For example, by the b_gen_per_ex rule, Blocks(C, $\mathcal{I}$, $\Box_{u_1}$) = $\bigcup$ CompoundBlocks(C, $\mathcal{I}$, $\Box_{u_1}$) ∪ SimpleBlocks where SimpleBlocks = ∅ because no component in C satisfies the example $\Box_{u_1} = \{(1, 2) \mapsto 1\}$. Constraints over CompoundBlocks(C, $\mathcal{I}$, $\Box_{u_1}$) are generated by applying the rules in a similar manner. For instance, by the b_proj rule, a constraint x.1 ∈ CompoundBlocks(C, $\mathcal{I}$, $\Box_{u_1}$) will be generated since x.1 ∈ Deduce(C, $\mathcal{I}$, $\Box_{u_1}$). Blocks(C, $\mathcal{I}$, $\Box_{u_4}$) will include a version space of a single expression add since add is a component satisfying the example. Blocks(C, $\mathcal{I}$, $\Box_{u_6}$) (where $\Box_{u_6} = \{(1, 2) \mapsto (1, 1)\}$) will include a version space of a tuple (Blocks(C, $\mathcal{I}$, $\Box_{u_1}$), Blocks(C, $\mathcal{I}$, $\Box_{u_1}$)) by the b_tuple rule.

When generating constraints, a cycle that leads to blocks of infinite length may occur. For example, we may generate the following two constraints: S(Blocks(C, $\mathcal{I}$, $\Box_{u_1}$)) ⊆ CompoundBlocks(C, $\mathcal{I}$, $\Box_u$)

and $\mathsf{S}^{-1}(\mathsf{Blocks}(\mathsf{C}, \mathcal{I}, \square_u)) \subseteq \mathsf{CompoundBlocks}(\mathsf{C}, \mathcal{I}, \square_{u_1})$ that can be used to generate blocks of form $\mathsf{S}(\mathsf{S}^{-1}(\mathsf{S}(\mathsf{S}^{-1}(\cdots))))$. In our implementation, we bound the maximum height of version spaces to avoid generating blocks of infinite length.

We can derive the final version space of blocks for $\square_u$ by finding the following smallest version space satisfying the above constraints.

$$\mathsf{Blocks}(\mathsf{C}, \mathcal{I}, \square_u) = \bigcup\{2, \mathsf{x.2}, \mathsf{S(x.1)}, (\mathsf{add}\ (\bigcup\{\mathsf{x.1}, \mathsf{S}^{-1}(\mathsf{x.2}), \cdots\}, \bigcup\{\mathsf{x.1}, \mathsf{S}^{-1}(\mathsf{x.2}), \cdots\})), \cdots\}.$$

4.5 Pruning Infeasible Hypotheses Using Blocks

Finally, in this section, we describe the BlockConsistent procedure that prunes infeasible hypotheses using the blocks generated by the BlockGen procedure.

Deriving Blocks from a Hypothesis by Unfolding We first describe how to obtain blocks from an open hypothesis which we want to determine the feasibility of by a technique we call unfolding. Suppose a currently considered hypothesis is $P = \mathsf{rec}\ \mathsf{f(x)} = e_{\text{body}}$. For each input example i in the top-level input-output example $\square_{in}$, we perform symbolic evaluation (interleaved with concrete evaluation) over e_{body} and obtain a block, which possibly contains holes. We call such a block possibly with holes (resp. without holes) an *open block* (resp. *closed block*). We formalize our symbolic evaluation via the transition relation $e \ \rightarrow_{P,i}\ e'$ induced by the target hypothesis P and the input i. The relation says that the expression e takes a single step to the expression e'. The transition relation is formally defined by the rules in Fig. 5. The rules U_CTOR, U_DTOR, U_TUPLE, and U_PROJ perform symbolic evaluation on the arguments of constructor, destructor, tuple, and projection expressions, respectively. U_APP_L and U_APP_R perform symbolic evaluation on the left and right hand sides of applications. The most notable part is the remaining two rules. U_MATCH for match expressions concretely evaluates the scrutinee e of a given match expression with input i. Then, a branch is chosen by the concrete value of the scrutinee. To obtain concrete values of scrutinees, we require scrutinees not to contain recursive calls to the target function, which is unknown yet. Therefore, any hypothesis containing a match expression that pattern matches on a recursive call to the target function (called *inside-out recursion* [Osera 2015]) will get stuck and thus will be determined to be infeasible. This means *Candidate generator* will never generate programs with inside-out recursion. However, such programs can still be synthesized by our algorithm as *Bottom-up enumerator* will eventually enumerate all programs. U_REC is a special rule for recursive calls. Any recursive call to the target function f is replaced by the body of the function where every occurrence of the parameter x is replaced by the argument expression. Note that there are no transition rules for variables and holes. That is, every variable and hole in the hypothesis remains unchanged.

Given the top-level input-output examples $\square_{in} = \bigcup_{1 \le j \le n}\{i_j \mapsto o_j\}$, the set of open blocks derivable from hypothesis $P = \mathsf{rec}\ \mathsf{f(x)} = e_{\text{body}}$ (denoted B_P) is defined as follows:

$$B_P = \{(i_j, o_j) \mapsto e \mid e_{\text{body}} \ \rightarrow^*_{P,i_j}\ e, 1 \le j \le n\}.$$

where $\rightarrow^*_{P,i_j}$ is the transitive closure of $\rightarrow_{P,i_j}$. That is, with each input example, we apply the transition rules till the end to obtain an open block.

Example 4.11. Recall the hypothesis P_4 in the overview example in Section 2.

P_4 = `rec f (x) = match x.1 with Z -> x.1 | S _ -> add (f (`S^{-1}`(x.1), x.2),`$\square_5$`)`

Using the rules in Fig. 5, we can derive an open block from P_4 with input example $i = (1, 2)$ as follows:

$$\frac{e_i \rightarrow_{P,i} e_i' \qquad 1 \le i \le k}{\kappa(e_1, \cdots, e_i, \cdots, e_k) \rightarrow_{P,i} \kappa(e_1, \cdots, e_i', \cdots, e_k)}\ \text{U\_CTOR} \qquad \frac{e \rightarrow_{P,i} e'}{\kappa^{-1}(e) \rightarrow_{P,i} \kappa^{-1}(e')}\ \text{U\_DTOR}$$

$$\frac{e_i \rightarrow_{P,i} e_i' \qquad 1 \le i \le k}{(e_1, \cdots, e_i, \cdots, e_k) \rightarrow_{P,i} (e_1, \cdots, e_i', \cdots, e_k)}\ \text{U\_TUPLE} \qquad \frac{e \rightarrow_{P,i} e'}{e.n \rightarrow_{P,i} e'.n}\ \text{U\_PROJ}$$

$$\frac{e_1 \rightarrow_{P,i} e_1'}{e_1\ e_2 \rightarrow_{P,i} e_1'\ e_2}\ \text{U\_APP\_L} \qquad \frac{e_2 \rightarrow_{P,i} e_2'}{e_1\ e_2 \rightarrow_{P,i} e_1\ e_2'}\ \text{U\_APP\_R}$$

$$\frac{\sigma[\mathsf{x} \mapsto i] \vdash e \Rightarrow \kappa_m\ \_ \qquad 1 \le m \le k}{\mathsf{match}\ e\ \mathsf{with}\ \overline{\kappa_j\ \_ \rightarrow e_j}^{k} \rightarrow_{P,i} e_m}\ \text{U\_MATCH} \qquad \frac{}{\mathsf{f}\ e \rightarrow_{P,i} e_{\text{body}}[e/\mathsf{x}]}\ \text{U\_REC}$$

Fig. 5. Rules for unfolding (symbolic evaluation interleaved with concrete evaluation) for deriving open blocks from P = rec $\mathsf{f(x)} = e_{\text{body}}$ with input i

$\mathtt{match\ x.1\ with\ Z\ \text{->}\ x.1\ |\ S\ \_\ \text{->}\ add\ (f\ (S^{-1}(x.1), x.2), \square_5)}$

$\rightarrow_{P_4,i}$ $\mathtt{add\ (f\ (S^{-1}(x.1), x.2), \square_5)}$ (By U_MATCH)

$\rightarrow_{P_4,i}$ $\mathtt{add\ (match\ S^{-1}(x.1)\ with\ Z\ \text{->}\ S^{-1}(x.1)\ |\ S\ \_\ \text{->}\ add\ (f\ (S^{-2}(x.1), x.2), \square_5),\ \square_5)}$
(By U_REC, U_TUPLE, and U_APP_R)

$\rightarrow_{P_4,i}$ $\mathtt{add\ (S^{-1}(x.1),\ \square_5)}$ (By U_MATCH, U_TUPLE, and U_APP_R)

Checking Feasibility of a Hypothesis We check the feasibility of a hypothesis P by checking if it is *block consistent* with respect to the set $\mathbf{B}$ of closed blocks from the BlockGen procedure, which is formally defined as follows:

Definition 4.12. Given the top-level input-output examples $\square_{in}$, a hypothesis P = rec $\mathsf{f(x)} = e$ is *block consistent* with respect to a set $\mathbf{B}$ of blocks if and only if

$$\forall i_j \mapsto o_j \in \square_{in}.\ B_P(i_j, o_j) \sim \mathbf{B}(i_j, o_j)$$

where $\sim$ is a binary relation over *Exp* and version spaces, which is defined in Fig. 6.

The relation $\sim$ relates an open block to a set of closed blocks. Specifically, for an open block e and a version space of closed blocks $\widetilde{e}$, $e \sim \widetilde{e}$ holds if we can obtain an expression in $\widetilde{e}$ by properly filling each occurrence of the holes in e. Checking if $e \sim \widetilde{e}$ resembles conventional syntactic matching between different expressions but with the following differences. Syntactic matching has as a goal to determine whether two expressions can be made equal by searching for a proper substitution from variables into expressions. For example, add (x, Z) can be matched with add (Z, Z) since we can substitute x with Z. On the other hand, in our method, not variables but only holes are targets for substitution. In addition, in contrast to syntactic matching that traverses two expressions, our method simultaneously traverses one expression and a version space to figure out if an open block can be matched with a closed block in the version space.

The first rule in Fig. 6 says that any hole can be matched with any expression in $\widetilde{e}$ as long as $\widetilde{e}$ is not empty. The other rules recursively traverse the version space $\widetilde{e}$ of blocks and check if e can be matched with any expression in $\widetilde{e}$.

Finally, the BlockConsistent procedure is defined as follows:

$$\mathsf{BlockConsistent}(P, \mathbf{B}, \square_{in}) = \begin{cases} \mathtt{true} & (\text{if } \forall i_j \mapsto o_j \in \square_{in}.\ B_P(i_j, o_j) \sim \mathbf{B}(i_j, o_j)) \\ \mathtt{false} & (\text{otherwise}) \end{cases}$$

$$\frac{}{\square_u \sim \widetilde{e}}\ \widetilde{e} \neq \emptyset \qquad \frac{\bigvee_{1 \le m \le k} e \sim \widetilde{e_m}}{e \sim \bigcup\{\widetilde{e_1}, \cdots, \widetilde{e_k}\}} \qquad \frac{\forall 1 \le m \le k.\ e_m \sim \widetilde{e_m}}{(e_1, \cdots, e_k) \sim (\widetilde{e_1}, \cdots, \widetilde{e_k})} \qquad \frac{\forall 1 \le m \le k.\ e_m \sim \widetilde{e_m}}{\kappa(e_1, \cdots, e_k) \sim \kappa(\widetilde{e_1}, \cdots, \widetilde{e_k})}$$

$$\frac{e = \kappa^{-1}(e') \quad e' \sim \widetilde{e}}{e \sim \kappa^{-1}(\widetilde{e})} \qquad \frac{e_1 \sim \widetilde{e_1} \quad e_2 \sim \widetilde{e_2}}{e_1\ e_2 \sim (\widetilde{e_1}\ \widetilde{e_2})} \qquad \frac{e \sim \widetilde{e}}{e.n \sim \widetilde{e}.n} \qquad \frac{}{e \sim \widetilde{e}}\ \widetilde{e} = e$$

Fig. 6. Matching rules for checking block consistency

Example 4.13. The following derivation tree shows how the open block in Example 4.11 can be matched with the version space $(\mathsf{add}\ (\bigcup\{\mathsf{Z}, \mathsf{S}^{-1}(\mathsf{x.1})\}, \bigcup\{\mathsf{x.2}, \mathsf{S}(\mathsf{x.1})\})$ using the rules in Fig. 6.

$$\frac{\frac{}{\mathsf{add} \sim \mathsf{add}} \qquad \frac{\frac{\frac{}{\mathsf{S}^{-1}(\mathsf{x.1}) \sim \mathsf{S}^{-1}(\mathsf{x.1})}}{\mathsf{S}^{-1}(\mathsf{x.1}) \sim \bigcup\{\mathsf{Z}, \mathsf{S}^{-1}(\mathsf{x.1})\}} \qquad \frac{}{\square_5 \sim \bigcup\{\mathsf{x.2}, \mathsf{S}(\mathsf{x.1})\}}}{(\mathsf{S}^{-1}(\mathsf{x.1}), \square_5) \sim (\bigcup\{\mathsf{Z}, \mathsf{S}^{-1}(\mathsf{x.1})\}, \bigcup\{\mathsf{x.2}, \mathsf{S}(\mathsf{x.1})\})}}{\mathsf{add}\ (\mathsf{S}^{-1}(\mathsf{x.1}), \square_5) \sim (\mathsf{add}\ (\bigcup\{\mathsf{Z}, \mathsf{S}^{-1}(\mathsf{x.1})\}, \bigcup\{\mathsf{x.2}, \mathsf{S}(\mathsf{x.1})\})}$$

As already mentioned in Section 2, the block-based pruning presented may be *unsound*; a valid open hypothesis that can be a solution in the future may be pruned by the block-based pruning. Such a situation may occur if *Block generator* is not able to generate closed blocks for the valid hypothesis due to a lack of components. However, as the component pool grows, such unsoundness may be mitigated.

Please recall that even though the block-based pruning is unsound, our algorithm finds a solution if exists by resorting to *Bottom-up enumerator* that will eventually generate a solution of finite size.

5 IMPLEMENTATION

In this section, we describe various implementation details of our synthesis algorithm.

5.1 Ensuring Termination of Block and Candidate Generation

In our implementation, to guarantee the termination of the BlockGen procedure and the Deduce procedure, we limit the maximum number of subsequent steps of deduction to a certain number. In other words, to avoid generating infinitely many open hypotheses from a given initial hypothesis, we permit the Deduce procedure to be terminated after a certain number of steps of applications of rules in Fig. 3. Because the BlockGen procedure relies on the Deduce procedure, this also guarantees termination of the BlockGen procedure. Note that despite this finitization, the search space is still infinite because there is no limit on the maximum component size (i.e., the component pool will keep growing until a solution is found).

5.2 Program Selection

In order to synthesize likely programs, we utilizes a cost function: the cost of each candidate program $P = \text{rec } \mathsf{f}(\mathsf{x}) = e$ is determined by the cost of its body e (denoted $C(e)$) which is a non-negative number. Costs of expressions satisfy the following constraints (some cases are omitted):

- $C(e_1\ e_2) > C(e_1) + C(e_2)$
- $C(\kappa(e_1, \cdots, e_k)) > \sum_{1 \le j \le k} C(e_j)$
- $C(x) = 0$
- $C(e.n) = C(e)$
- $C(\kappa^{-1}(e)) = C(e)$

Intuitively, we penalize the use of constructors (thereby constants) and prioritize the use of variables, destructors, and projections. The reason for this is that they are used to extract subcomponents of constructors, which are essentially the same as variables bound by patterns in match expressions. For example, consider the solution P_{sol} of the overview example problem in Section 2.

rec mul (x : nat, y : nat) : nat = match x with Z -> Z | S _ -> add (mul (S^{-1}(x), y), y)

This program can be written as the following program by introducing a new variable x' bound by the pattern of S.

```
rec mul (x : nat, y : nat) : nat = match x with Z -> Z | S x' -> add (mul (x', y), y)
```

Note that S^{-1}(x) and x' play the same role. Therefore, destructors and projections can be understood as variables in many cases, and prioritizing variables has been proved to be a good heuristic to avoid overfitting [Feser et al. 2015; Gulwani 2011]. Lastly, in case of a tie, we pick a smaller program in terms of AST size. This heuristic has also been popularly used in the majority of previous approaches [Albarghouthi et al. 2013; Feser et al. 2015; Miltner et al. 2022; Wang et al. 2017].

5.3 Checking Block Consistency

We describe implementation details for improving the pruning power of the BlockConsistent procedure. In Algo. 1, when choosing a hole in a hypothesis of a match expression, we prefer holes for base cases to ones for inductive cases. By filling holes for base cases first, we can effectively prune infeasible recursive hypotheses. For example, suppose we encounter the following hypothesis while synthesizing mul in Section 2.

P_1 = rec mul (x : nat, y : nat) : nat = match x with Z -> $\square_1$ | S _ -> mul (S^{-1}(x), $\square_2$)

Suppose we first fill the hole $\square_2$ with y, obtaining the following hypothesis.

P_2 = rec mul (x : nat, y : nat) : nat = match x with Z -> $\square_1$ | S _ -> mul (S^{-1}(x), y)

Note that this hypothesis cannot become a solution no matter what expression we put in the remaining hole. To check block consistency, for every input, we will obtain the open block $\square_1$ as a result of our symbolic evaluation. Although the hypothesis P_2 is infeasible, because a hole can be matched with any expression according to the rules in Fig. 6, the hypothesis will be determined to be block consistent with respect to any set of blocks, and will not be pruned.

Now, suppose we first fill the hole $\square_1$ in P_1 with x, obtaining the following hypothesis.

P_3 = rec mul (x : nat, y : nat) : nat = match x with Z -> x | S _ -> mul (S^{-1}(x), $\square_2$)

Note that this hypothesis also cannot become a solution no matter what expression we put in the remaining hole. For every input, we will obtain a closed block x as a result of our symbolic evaluation. Because x is not included in the blocks for the example $(1, 2) \mapsto 2$, the hypothesis P_3 is determined to be block inconsistent and will be pruned.

5.4 Optimizations

We also utilize a few standard optimizations in prior work. For ease of presentation, we have presented as if we do not type-check any of the expressions during the search. However, in our implementation, we perform type-based pruning to generate only well-typed expressions, similarly to prior work [Feser et al. 2015; Osera and Zdancewic 2015]. We also generate expressions in β-normal η-long form as done in prior work [Frankle et al. 2016; Lubin et al. 2020; Osera and Zdancewic 2015]. Also, we apply constructor/destructor simplification [Lubin et al. 2020] to avoid generating unnecessarily long programs containing sub-expressions of forms $\kappa(\kappa^{-1}\ (\cdots))$ or $\kappa^{-1}(\kappa\ (\cdots))$ for any constructor κ.

5.5 Using Another Version of the D_EXTCALL Rule

As described in Section 4.3, the D_EXTCALL rule in Fig. 3 may be unsound (i.e., some of generated holes cannot be filled with any expression). This deduction unsoundness may lead to a scalability issue by generating too many unsatisfiable holes if the number of examples is greater than a certain threshold. In such a case, we use the following two rules instead of the D_EXTCALL rule.

$$\frac{\begin{array}{c}\square_u = \bigcup_{1\le j\le n}\{i_j \mapsto o_j\} \quad e_1, e_2 \in \mathbf{C} \quad \forall 1 \le j \le n.\ \sigma[\mathsf{x}\mapsto i_j] \vdash e_1 \Rightarrow g_j \\ \forall 1 \le j \le n.\ \sigma[\mathsf{x}\mapsto i_j] \vdash e_2 \Rightarrow v_j \quad \forall 1 \le j \le n.\ {g_j}^{-1}(o_j) = v_j \in \mathcal{I}\end{array}}{e_1\ e_2 \in \mathsf{Deduce}(\mathbf{C}, \mathcal{I}, \square_u)}\ \textsc{d\_extcall1}$$

$$\frac{\begin{array}{c}\square_u = \bigcup_{1\le j\le n}\{i_j \mapsto o_j\} \quad e_1, e_2 \in \mathbf{C} \quad \forall 1 \le j \le n.\ \sigma[\mathsf{x}\mapsto i_j] \vdash e_1 \Rightarrow g_j \\ e_2 \text{ contains recursive calls.} \quad \forall 1 \le j \le n. \exists v.\ {g_j}^{-1}(o_j) = v \in \mathcal{I}\end{array}}{e_1\ e_2 \in \mathsf{Deduce}(\mathbf{C}, \mathcal{I}, \square_u)}\ \textsc{d\_extcall2}$$

Both rules use components to generate closed expressions without any holes. The difference between the two rules is in whether a component for the argument part contains recursive calls. The D_EXTCALL2 rule considers every component cotaining recursive calls to be the potential argument of a library function call. This is based on an optimistic assumption that any recursive expression may be a proper argument (like the D_REC rule in Fig. 3). These rules do not generate any unsatisfiable holes, but the search space explored in a single iteration of the main loop (lines 2 – 25 in Algo. 1) is smaller than the case where the D_EXTCALL rule is used.

5.6 Finding a Solution vs. All Solutions

We allow the user to choose between finding all possible solutions synthesizable using some set of components and picking the best one, and stopping the search as soon as a solution is found. This allows the user to find a good balance between speed of synthesis and accuracy (i.e., the possibility of generating intended programs). Finding all solutions and picking the best one only requires a slight modification in Algo. 1 as follows: even if a solution is found on line 11, the main loop continues to explore the search space until the queue becomes empty. Then, we pick the best one among the multiple solutions found so far. In addition, we can further expedite the process of finding a single solution with a slight modification in Fig. 3. Instead of including all components consistent with a given set of input-output examples in the D_COMPONENT rule, we just include a single component whose score is the best among the satisfying components.

6 EVALUATION

We have implemented our approach in a tool Trio. Trio consists of about 4K lines of OCaml code. We evaluate Trio on synthesis tasks used in prior work and on new tasks collected from an online tutorial. We aim to answer the following research questions:

- **RQ1**: How does Trio perform on various synthesis tasks?
- **RQ2**: How does Trio compare with existing techniques for recursive program synthesis?
- **RQ3**: How effective are block-based pruning and library sampling for accelerating synthesis?

All of our experiments were conducted on a 2.0 GHz Intel Core i5 processor with 16 GB of memory running macOS Big Sur. We set the timeout limit to 120 seconds for each synthesis task.

6.1 Experimental Setup

Benchmarks. We use 60 recursive functional programs. 45 out of 60 have been used to evaluate prior work [Lubin et al. 2020; Miltner et al. 2022; Osera and Zdancewic 2015]. The remaining 15 programs are from the exercises in the official OCaml online tutorial and their slight variants. The details can be found in Table 1.

For these benchmarks, we consider the following two classes of specifications to evaluate Trio over different specifications.

Table 1. List of new 15 benchmarks collected from the exercises in the official OCaml online tutorial (https://ocaml.org/problems) and their variants.

	Name	External Operators	Description
Arithmetic	nat_mul	add	Multiplication of two natural numbers.
	nat_sub		Subtraction of two natural numbers.
	expr_boolean	not, and, or	Logical expression evaluator.
	expr	add, mul	Calculator using addition and multiplication operators.
	expr_sub	add, mul, sub	Calculator using addition, subtraction, and multiplication operators.
	expr_div	add, mul, sub, div	Calculator using addition, subtraction, multiplication, and division operator.
Lists	list_dropeven	is_even	Drop the even number(s) in a list.
	list_last2		Return the last two elements in a list.
	list_make		Return the 0 padded list of length *n*
	list_range	compare	Return the sequence of the given numbers in descending order.
Trees	tree_height	compare, max	Return the height of a tree.
	tree_balanced	compare, max, tree_height, and	Check whether a tree is height-balanced.
	tree_lastleft		Return the node at the left end of the tree.
	tree_notexist	compare, and	Check if a number is in a tree
	tree_sum	add	Return the sum of numbers in a tree.

- **IO**: We use input-output examples written by developers of SMYTH [Lubin et al. 2020] and BURST [Miltner et al. 2022].
- **Ref**: For 45 benchmarks, we use reference implementations from prior work written by developers of BURST [Miltner et al. 2022]. For the other 15 benchmarks, we use reference implementations from the official OCaml online and the ones written by us.

Baselines. We compare TRIO to state-of-the-art tools for synthesizing recursive functional programs. SMYTH [Lubin et al. 2020] performs top-down synthesis from input-output examples. It performs partial evaluation to propagate constraints from partial programs to the remaining holes. BURST [Miltner et al. 2022] performs bottom-up synthesis from input-output examples or logical specifications. Neither of them requires trace-complete specifications. We aim to confirm the benefits of our bidirectional search strategy by comparing TRIO against the top-down synthesizer SMYTH and the bottom-up synthesizer BURST.

6.2 Effectiveness of TRIO for Input-Output Examples

We evaluate TRIO on synthesis problems with **IO** specifications. The initial component size *n* for TRIO is set to be 6. For each instance, we measure the running time of TRIO and the size of the synthesized program.

The results are summarized in Table. 2. The column "Correct" shows if the synthesized program is the one intended by the user. We manually checked if the synthesized program is semantically equivalent to the known solution for each problem.

TRIO outperforms the other baselines in terms of both the number of solved problems and synthesis time. TRIO can synthesize 59 out of 60 problems, with an average time of 1.9 seconds. On the other hand, BURST and SMYTH can synthesize 50 problems, with average times of 2.1 and 4.4 seconds, respectively. In addition, TRIO is the fastest tool in 38 problems, whereas BURST and SMYTH are the fastest tools in 15 and 32 problems, respectively. [4]

For every problem, the time taken for synthesizing blocks and constructing inverse maps is negligible (usually less than a second).

We observe BURST and SMYTH occasionally take a large amount of memory, whereas TRIO only requires a small amount of memory. While solving all of the tasks, the peak memory usage of BURST is 1.3 GB and that of SMYTH is 1.2 GB. On the other hand, TRIO only requires 67 MB even in

[4] If there are ties in a synthesis problem, all tools with the same synthesis time are considered to be the fastest.

Table 2. Results for the **IO** benchmark suite (with 12 easy problems omitted), where "Time" gives synthesis time in seconds, and "Size" shows the size of the synthesized program (measured by number of AST nodes). Synthesis time of the fastest tool for each problem is highlighted in bold.

Benchmark	**Trio**			**Burst**			**SMyth**		
	Time(s)	Size	Correct	Time(s)	Size	Correct	Time(s)	Size	Correct
bool_band	**0.02**	14	✓	**0.02**	14	✓	0.03	23	✓
list_append	0.04	44	✗	0.15	31	✓	**0.03**	33	✓
list_compress	**0.13**	56	✓	0.85	58	✓	Timeout	N/A	N/A
list_concat	0.04	19	✓	0.03	19	✓	**0.02**	17	✓
list_drop	**0.03**	31	✓	1.52	24	✓	0.04	32	✓
list_even_parity	**0.02**	25	✓	0.03	23	✓	0.03	22	✓
list_filter	**0.03**	47	✗	0.33	46	✗	0.07	44	✗
list_fold	**0.07**	41	✗	0.13	43	✗	1.24	49	✗
list_map	**0.03**	37	✓	**0.03**	37	✓	0.38	51	✗
list_nth	**0.04**	33	✓	1.93	33	✓	**0.04**	34	✓
list_pairwise_swap	**0.03**	38	✓	0.04	38	✓	0.04	27	✓
list_rev_append	0.09	35	✗	1.03	22	✓	**0.05**	20	✓
list_rev_fold	**0.02**	10	✓	0.03	10	✓	0.04	15	✓
list_rev_snoc	0.04	18	✓	1.33	18	✓	**0.02**	16	✓
list_rev_tailcall	**0.02**	37	✗	0.18	43	✗	**0.02**	33	✓
list_snoc	**0.03**	36	✓	1.93	36	✓	**0.03**	37	✓
list_sort_sorted_insert	0.04	18	✓	0.07	18	✓	**0.02**	16	✓
list_sorted_insert	**0.13**	60	✗	Timeout	N/A	N/A	1.44	55	✓
list_stutter	**0.02**	23	✓	3.45	23	✓	**0.02**	21	✓
list_sum	**0.02**	10	✓	0.07	10	✓	0.03	10	✓
list_take	**0.03**	38	✓	Timeout	N/A	N/A	**0.03**	38	✓
nat_add	0.09	22	✓	0.03	22	✓	**0.02**	27	✓
nat_iseven	**0.02**	16	✓	0.03	16	✓	**0.02**	13	✓
nat_max	0.16	23	✓	0.18	23	✓	**0.06**	34	✓
tree_binsert	**0.61**	87	✓	2.44	87	✓	Timeout	N/A	N/A
tree_collect_leaves	0.07	27	✓	0.45	27	✓	**0.04**	24	✓
tree_count_leaves	**0.07**	22	✓	0.18	22	✓	0.46	25	✓
tree_count_nodes	0.17	22	✓	**0.07**	22	✓	0.16	20	✓
tree_inorder	0.12	27	✓	7.89	27	✓	**0.06**	24	✓
tree_map	**0.06**	49	✓	0.40	49	✓	0.92	61	✓
tree_nodes_at_level	**0.30**	47	✓	32.63	47	✓	Timeout	N/A	N/A
tree_postorder	**0.48**	32	✓	2.86	32	✓	Timeout	N/A	N/A
tree_preorder	0.13	27	✓	0.10	27	✓	**0.09**	24	✓
nat_mul	**1.16**	27	✓	Timeout	N/A	N/A	105.53	31	✓
nat_sub	**0.09**	29	✓	7.78	29	✓	0.04	30	✓
expr_boolean	16.4	59	✓	**0.61**	52	✗	Timeout	N/A	N/A
expr	**0.77**	36	✓	Timeout	N/A	N/A	Timeout	N/A	N/A
expr_sub	13.81	51	✓	Timeout	N/A	N/A	**10.79**	58	✗
expr_div	Timeout	N/A	N/A	Timeout	N/A	N/A	Timeout	N/A	N/A
list_dropeven	**0.03**	28	✓	Timeout	N/A	N/A	**0.03**	25	✓
list_last2	**0.03**	40	✓	0.10	39	✓	0.07	29	✓
list_make	0.02	14	✓	0.83	14	✓	**0.01**	13	✓
list_range	**0.18**	60	✓	Timeout	N/A	N/A	Timeout	N/A	N/A
tree_balanced	**75.40**	45	✗	Timeout	N/A	N/A	Timeout	N/A	N/A
tree_height	**4.48**	36	✗	Timeout	N/A	N/A	96.62	29	✗
tree_lastleft	0.24	21	✓	0.03	21	✓	**0.02**	18	✓
tree_notexist	**0.42**	79	✓	0.78	79	✓	Timeout	N/A	N/A
tree_sum	0.47	28	✓	34.01	28	✓	**0.44**	25	✓
# Solved (# correct)	**59 (51)**			**50 (46)**			**50 (45)**		
# Timeout	**1**			**10**			**10**		

the worst case. On average, Burst and SMyth use 80 and 41 MB of memory respectively, whereas Trio uses 22 MB. Thus we can conclude Trio is more memory efficient than the other baselines.

Analysis of Overfitting. We manually inspect the programs synthesized by the three tools to investigate how they are prone to overfitting. 51 out of 59 programs (86%) synthesized by Trio are the intended ones. 46 out of 50 programs (92%) and 45 out of 50 programs (90%) synthesized by Burst and SMyth are the intended ones, respectively. Therefore, all the tools are roughly equal in terms of solution quality.

We can mitigate overfitting by making Trio find all solutions that can be found with a current set of components and choose the best one according to the cost described in Section 5.2. For the 8 problems for which Trio synthesizes unintended programs, if Trio is configured to find all solutions and pick the best one, it could find the desired programs for 5 problems at the cost of reasonable overhead (ranging from a second to a few minutes). However, Trio cannot find the desired solution of `list_rev_tailcall` because of our method for guaranteeing the termination of synthesized programs (Section 5.1). `list_rev_tailcall` requires tail-recursive calls but our current termination checker based on the default value order does not permit tail-recursive calls. More specifically, `list_rev_tailcall` requires a recursive call on ([2],[1]) for input ([1;2],[]), but our value ordering does not consider ([2],[1]) to be strictly smaller than ([1;2],[]). For the same reason, Burst also fails to find the solution. This issue can be resolved by using a more sophisticated termination checking method in the future. In the case of `list_rev_append`, the specification is not constraining enough for finding the solution. In the experiment with reference implementations based on CEGIS where infinitely many input-output examples can be provided, we confirm that Trio successfully finds the desired solution.

Failure Analysis. We analyze one problem for which Trio timed out. The problem `expr_div` is hard in that it requires complex pattern matching involving many external operators. It concerns synthesizing a simple calculator with addition, subtraction, multiplication, and division. The specification is given as follows:

```
type nat = Z | S of nat
type expr = NAT of nat | ADD of expr and expr | SUB of expr and expr
           | MUL of expr and expr | DIV of expr and expr
rec add (x : nat, y : nat) : nat = ...     rec sub (x : nat, y : nat) : nat = ...
rec mul (x : nat, y : nat) : nat = ...     rec div (x : nat, y : nat) : nat = ...
rec eval (x : nat, y : nat) : nat = □_in
```

where $\square_{in}$ is the set of input-output examples $\{\mathtt{NAT}\ 1 \mapsto 1, \mathtt{ADD}(\mathtt{NAT}\ 1, \mathtt{NAT}\ 2) \mapsto 3, \cdots\}$, and `add`, `sub`, `mul`, and `div` are the external operators. Finding the following solution is non-trivial because there are extremely many possible combinations of recursive calls, external operators, and case matching.

```
rec eval (x : expr) : nat =
  match x with NAT n -> n
  | ADD (x1, x2) -> add (eval x1, eval x2)   | SUB (x1, x2) -> sub (eval x1, eval x2)
  | MUL (x1, x2) -> mul (eval x1, eval x2)   | DIV (x1, x2) -> div (eval x1, eval x2)
```

However, Trio can find the solution if the specification is restricted to addition, subtraction, and multiplication (which is `expr_sub`) in 16 seconds. On the other hand, the other baselines fail to solve all the problems that concern synthesizing calculators (i.e., `expr`, `expr_sub`, `expr_div`).

Summary of Results. When synthesizing recursive programs from input-output examples, Trio outperforms state-of-the-art baseline tools in terms of both synthesis time and memory usage. Also, Trio solves harder synthesis problems beyond the reach of the baselines.

6.3 Effectiveness of Trio for Reference Implementations

In this section, we evaluate Trio on synthesis problems with Ref specifications. We follow the same evaluation procedure as the evaluation of Burst [Miltner et al. 2022] for **Ref** specifications. The authors of Burst integrated Burst and SMyth into a CEGIS loop and, for each candidate program proposed by each tool, they use the verifier to determine whether the candidate is semantically equivalent to the reference implementation. If not, a new input-output example comprising a counter-example input generated by the verifier and its corresponding output is added [5]. This process is repeated until the desired program is found.

The goal of this experiment is to confirm how the tools deal with the random examples generated by the verifier, rather than hand-crafted examples.

The results are summarized in Table 3. The column "# Iters" shows the number of CEGIS iterations required until a solution is found. Trio also outperforms the other baselines in terms of both the number of solved instances and synthesis time. Trio can synthesize 57 instances, with an average time of 3.9 seconds and an average number of CEGIS iterations of 5.1. Burst can synthesize 54 instances, with an average time of 6.7 seconds and an average number of CEGIS iterations of 5. SMyth can synthesize 39 instances, with an average time of 5.3 seconds and an average number of CEGIS iterations of 4.6.

Overall these results suggest that Trio can deal better with random examples generated by the verifier compared to the other baselines.

Failure Analysis. The reason for the timeout on `list_rev_tailcall` and `expr_div` is same as the reason already described in the previous experiment with IO specifications. The solution to the problem `tree_balanced` requires the use of deeply nested pattern matching in combination with four external operators. Therefore, it is challenging to solve the problem for all three tools.

Summary of Results. Also when synthesizing recursive programs from reference implementations, Trio outperforms the other baselines in terms of both the number of solved instances and synthesis time. The results suggest the Trio's robustness to randomly given examples.

6.4 Ablation Study for Block-Based Pruning and Library Sampling

We now evaluate the effectiveness of the block-based pruning and library sampling techniques used by Trio. For this purpose, we compare the performance of four variants of Trio, each using a different combination: Trio with block-based pruning and library sampling, Trio^B only with block-based pruning, Trio^L only with library sampling, and Trio^{--} with both techniques disabled.

Table 4 summarizes the results of this ablation study (more detailed results can be found in cactus plots in Fig. 7). For each variant of Trio, we report the number of solved benchmarks with the **IO** and **Ref** specifications respectively. In this experiment, we only consider the 15 newly added benchmarks because we realize the other 45 benchmarks from prior work are easy, so that they can be quickly solved by all the variants of Trio. As can be seen in the table, Trio with the two techniques solves more benchmarks than the other three variants. Trio solved 93% of the new benchmarks with IO specifications, whereas Trio^{--} could solve only 53% of the benchmarks. Such

[5] The authors of Burst use bounded testing instead of verification, and manually checked the semantic equivalence between the generated programs and the reference implementations. We use the same method by reusing the artifacts of Burst.

Table 3. Results for the **Ref** benchmark suite where "# Iter" shows the number of CEGIS iterations.

Benchmark	**Trio**			**Burst**			**SMyth**		
	Time(s)	Size	# Iter	Time(s)	Size	# Iter	Time(s)	Size	# Iter
bool_band	**0.02**	14	3	**0.02**	14	3	**0.02**	23	3
list_append	0.52	31	6	**0.37**	31	7	0.43	33	8
list_compress	**1.11**	56	10	1.14	56	9	Timeout	N/A	N/A
list_concat	1.20	19	5	0.90	19	4	**0.88**	17	4
list_drop	**0.48**	31	5	0.59	31	6	Timeout	N/A	N/A
list_even_parity	**0.12**	25	5	**0.12**	23	6	0.14	22	6
list_filter	1.02	56	8	**0.80**	56	8	Timeout	N/A	N/A
list_fold	**1.05**	42	6	26.37	42	7	Timeout	N/A	N/A
list_map	0.72	37	6	**0.61**	37	4	Timeout	N/A	N/A
list_nth	0.41	33	6	**0.35**	33	7	0.42	34	5
list_pairwise_swap	0.41	38	7	**0.34**	38	6	Timeout	N/A	N/A
list_rev_append	**0.96**	22	5	10.31	22	5	Timeout	N/A	N/A
list_rev_fold	0.72	10	3	**0.64**	10	2	Timeout	N/A	N/A
list_rev_snoc	**1.02**	18	5	4.71	18	4	Timeout	N/A	N/A
list_rev_tailcall	Timeout	N/A	N/A	Timeout	N/A	N/A	**0.44**	33	9
list_snoc	**0.36**	36	3	0.42	36	3	0.47	36	7
list_sort_sorted_insert	1.32	18	6	1.49	18	5	**1.31**	16	6
list_sorted_insert	**0.71**	60	6	0.96	60	6	12.56	69	12
list_stutter	**0.31**	23	3	0.98	23	3	0.38	21	4
list_sum	0.72	10	2	**0.67**	10	2	**0.67**	10	2
list_take	**0.32**	38	8	12.65	38	9	0.42	38	7
nat_add	0.14	22	5	**0.06**	22	6	**0.06**	27	6
nat_iseven	**0.02**	16	4	**0.02**	16	4	**0.02**	13	4
nat_max	0.23	23	5	7.00	23	5	**0.13**	34	7
tree_binsert	**6.59**	87	6	16.51	87	8	Timeout	N/A	N/A
tree_collect_leaves	5.35	27	4	6.11	27	5	**5.20**	24	5
tree_count_leaves	**5.34**	22	4	5.89	22	4	5.67	25	4
tree_count_nodes	5.36	22	4	5.41	22	4	**5.33**	20	4
tree_inorder	7.69	27	4	10.10	27	5	**7.65**	24	6
tree_map	5.78	49	6	**5.66**	49	5	Timeout	N/A	N/A
tree_nodes_at_level	**4.02**	47	7	Timeout	N/A	N/A	21.99	43	5
tree_postorder	**9.47**	32	6	15.48	32	8	Timeout	N/A	N/A
tree_preorder	8.92	27	6	8.13	27	4	**7.75**	24	5
nat_mul	**0.94**	27	6	69.78	27	8	7.50	31	7
nat_sub	**0.25**	29	6	1.10	29	7	Timeout	N/A	N/A
expr_boolean	42.95	59	13	**6.63**	59	21	Timeout	N/A	N/A
expr	**4.41**	36	9	Timeout	N/A	N/A	Timeout	N/A	N/A
expr_sub	**62.76**	51	12	Timeout	N/A	N/A	Timeout	N/A	N/A
expr_div	Timeout	N/A	N/A	Timeout	N/A	N/A	Timeout	N/A	N/A
list_dropeven	**0.35**	28	6	**0.35**	28	6	Timeout	N/A	N/A
list_last2	0.66	40	7	1.74	39	5	**0.64**	29	6
list_make	**0.02**	14	3	0.03	14	3	**0.02**	13	3
list_range	**0.37**	49	7	96.25	49	6	Timeout	N/A	N/A
tree_balanced	Timeout	N/A	N/A	Timeout	N/A	N/A	Timeout	N/A	N/A
tree_height	9.62	23	6	**7.84**	22	6	9.14	20	5
tree_lastleft	8.49	21	6	7.70	21	4	**7.03**	18	5
tree_notexist	**7.78**	79	8	12.05	79	9	Timeout	N/A	N/A
tree_sum	9.78	28	5	9.66	28	5	**7.42**	25	5
# Solved	**57**			**54**			**39**		
# Timeout	**3**			**6**			**21**		

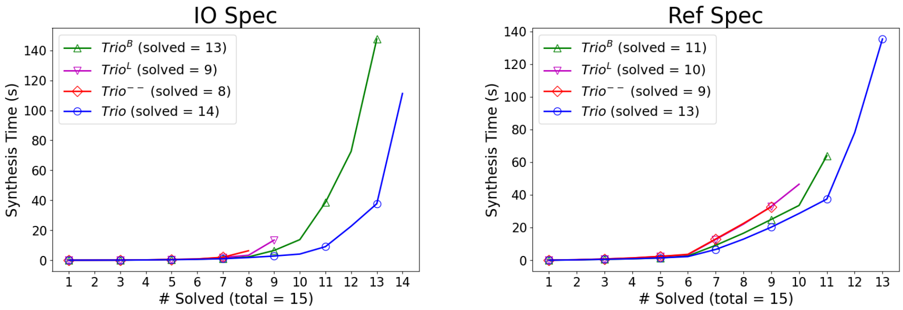

Fig. 7. Comparison of different variants of Trio.

Table 4. Number of instances that can be solved by four variants of Trio among **15** newly added benchmarks

	Trio	Trio^B	Trio^L	Trio^{--}
# Solved (**IO** spec.)	14 (**93%**)	13 (**87%**)	9 (**60%**)	8 (**53%**)
# Solved (**Ref** spec.)	13 (**87%**)	11 (**73%**)	10 (**66%**)	9 (**60%**)

a trend can also be observed in the reference implementation experiment. We notice the efficacy of block-based pruning is higher than that of library sampling because the difference between Trio^{--} and Trio^B is more significant than the difference between Trio^{--} and Trio^L.

6.5 Benefits of Our Method Compared to Prior Work

In this section, we analyze why our method outperforms the previous methods. As a representative example, we investigate how the tools work for a simpler version of the expr benchmark where Trio can quickly find the solution in contrast to the other baselines.

```
type nat = Z | S of nat
type expr = NAT of nat | ADD of expr and expr
rec add (x : nat, y : nat) : nat = ...      rec mul (x : nat, y : nat) : nat = ...
rec eval (x : nat, y : nat) : nat = □_in
```

where $\square_{in} = \{i_1 \mapsto 1, i_2 \mapsto 4, i_3 \mapsto 7\}$, $i_1 = \text{NAT } 1$, $i_2 = \text{ADD}(\text{NAT } 3, \text{NAT } 1)$, and $i_3 = \text{ADD}(\text{NAT } 4, \text{NAT } 3)$. The solution in our language (depicted in Fig. 2) is as follows:

```
  rec eval (x : expr) : nat =
     match x with NAT _ -> NAT^-1(x) | ADD _ -> add (eval ADD^-1(x).1, eval ADD^-1(x).2)
```

Comparison to SMyth. Similarly to our method, SMyth explores the search space by performing top-down propagation. There are two major differences between SMyth and our method. First, whenever a hole is filled with some expression, SMyth "updates" the other remaining holes according to the hole filling, so that the holes are more likely to be filled with the correct expressions. Second, SMyth solely relies on a top-down search strategy without any bottom-up search. We observe these two differences are the main reasons why SMyth fails to solve the benchmark. Consider the following hypothesis generated by SMyth during the search.

```
  P_1 = rec eval (x : expr) : nat = match x with NAT _ -> □_1 | ADD _ -> □_2
```

where $\square_1 = \{i_1 \mapsto 1\}$, $\square_2 = \{i_2 \mapsto 4, i_3 \mapsto 7\}$. SMyth further refines the hypothesis P_1 by filling the hole $\square_2$ with a recursive call to eval. Like Trio, SMyth enumerates structurally-decreasing recursive calls to guarantee the termination of synthesized programs. Suppose the following hypothesis is generated.

```
$P_2$ = rec eval (x : expr) : nat = match x with NAT _ -> $\square_1$ | ADD _ -> eval ADD$^{-1}$(x).1
```

Obviously, the hypothesis P_2 is not desired because it cannot be the solution. However, SMyth cannot detect this problem for the following reason. As hole $\square_2$ is filled, SMyth updates the other remaining hole. SMyth generates the following hypothesis.

```
$P_3$ = rec eval (x : expr) : nat = match x with NAT _ -> $\square_3$ | ADD _ -> eval ADD$^{-1}$(x).1
```

where $\square_3 = \square_1 \cup \{\mathsf{NAT}\ 3 \mapsto 4, \mathsf{NAT}\ 4 \mapsto 7\}$. The additional examples $\{\mathsf{NAT}\ 3 \mapsto 4, \mathsf{NAT}\ 4 \mapsto 7\}$ are originated from the original examples $\{i_2 \mapsto 4, i_3 \mapsto 7\}$ and partial evaluation of P_2 with the input examples. SMyth refines the hole $\square_3$ by generating the following hypothesis.

```
$P_4$ = rec eval (x : expr) : nat =
    match x with NAT _ ->
      match S$^{-1}$(x) with Z -> $\square_4$ | S _ -> $\square_5$
    | ADD _ -> eval ADD$^{-1}$(x).1
```

where $\square_4 = \{i_1 \mapsto 1\}$ and $\square_5 = \{\mathsf{NAT}\ 3 \mapsto 4, \mathsf{NAT}\ 4 \mapsto 7\}$. SMyth keeps refining this hypothesis, which is fruitless. In summary, SMyth's updating holes by partial evaluation sometimes makes the search more difficult.

On the other hand, Trio can quickly identify the infeasibility of P_2 as follows: Trio does not update the other remaining hole $\square_1$ after filling $\square_2$. Then, the hole $\square_1$ can be easily filled with NAT^{-1}(x), generating the following program, which can be easily proved to be infeasible by concrete evaluation.

```
$P_3'$ = rec eval (x : expr) : nat =
    match x with NAT _ -> NAT$^{-1}$(x) | ADD _ -> eval ADD$^{-1}$(x).1
```

In addition, we note that another source of inefficiency of SMyth is that it redundantly generates many semantically equivalent hypotheses. For instance, the followings are some of hypotheses generated by SMyth by filling $\square_1$ in P_1 with different expressions. The more library functions are usable, the more redundant hypotheses are generated.

```
match x with NAT _ -> NAT$^{-1}$(x)                | ADD _ -> $\square_2$
match x with NAT _ -> add (NAT$^{-1}$(x), 0)       | ADD _ -> $\square_2$
match x with NAT _ -> 1                       | ADD _ -> $\square_2$
match x with NAT _ -> add (1, (mul (0, 0))) | ADD _ -> $\square_2$
```

Note that the first two hypotheses and the last two hypotheses are semantically equivalent respectively. However, Trio avoids generating such redundant hypotheses because *Bottom-up enumerator* exploits observational equivalence to avoid generating multiple components of the same behaviors.

Comparison to Burst. Burst performs bottom-up synthesis with *angelic execution*. It first synthesizes a program assuming any recursive calls to the function being synthesized *angelically* behave to make the program correct. Then, it checks if the assumptions made in the previous step are correct. If they are, the solution is found. Otherwise, those assumptions are refuted and never made again by being added to the list of *anti-specifications*. This process is repeated, progressively strengthening the specification of the target function and eventually leading to a solution.

Burst fails to find the solution because it runs into extensive backtracking (i.e., too many steps of specification strengthening). Specifically, given the specification of the target function eval $i_1 = 1 \land$ eval $i_2 = 4 \land$ eval $i_3 = 7$ (which is from the three input-output examples), Burst first enumerates the following candidate program.

```
rec eval (x : expr) : nat = match x with NAT _ -> NAT^-1(x) | ADD _ -> eval ADD^-1(x).1
```

Obviously, the above program does not satisfy the second and third input-output examples. However, at this stage, Burst generates a program assuming any recursive call to eval can return anything to satisfy the constraints. The above program is generated by assuming eval (NAT 3) $=$ $4 \land$ eval (NAT 4) $= 7$. Burst then checks if this assumption is correct. It clearly does not because eval (NAT 3) $= 3$ and eval (NAT 4) $= 4$. Then, it re-attempts synthesis with the following strengthened specification.

$$\texttt{eval}\ i_1 = 1 \land \texttt{eval}\ i_2 = 4 \land \texttt{eval}\ i_3 = 7 \land \texttt{eval}\ (\mathsf{NAT}\ 3) = 4 \land \texttt{eval}\ (\mathsf{NAT}\ 4) = 7.$$

After the search within a bounded space, Burst fails to find a program that satisfies the strengthened specification. It concludes that the assumption made in the previous step is incorrect. Then, it adds the negation of the assumption (i.e., $\neg$(eval (NAT 3) $= 4 \land$ eval (NAT 4) $= 7$)) into the list of anti-specifications, searches for a program that does not violate the anti-specifications and generates the following program.

```
rec eval (x : expr) : nat =
    match x with NAT _ -> NAT^-1(x) | ADD _ -> S (S (eval ADD^-1(x).1))
```

Obviously, this program also does not satisfy the original specification. However, it does not violate the anti-specification because the above program is correct assuming eval (NAT 3) $=$ $2 \land$ eval (NAT 4) $= 5$. Then, it re-attempts synthesis with the following strengthened specification.

$$\texttt{eval}\ i_1 = 1 \land \texttt{eval}\ i_2 = 4 \land \texttt{eval}\ i_3 = 7 \land \neg(\texttt{eval}\ (\mathsf{NAT}\ 3) = 4 \land \texttt{eval}\ (\mathsf{NAT}\ 4) = 7)$$
$$\land \texttt{eval}\ (\mathsf{NAT}\ 3) = 2 \land \texttt{eval}\ (\mathsf{NAT}\ 4) = 5$$

Again, after a bounded search, Burst fails to find a program that satisfies the strengthened specification. It concludes that the assumption made in the previous step is incorrect. Then, Burst refutes the assumption, increasing the list of anti-specifications. In this manner, Burst initially overapproximates the specification of the target function and then refines it by repeatedly adding anti-specifications. However, because the space of possible anti-specifications is too large, this method is not effective.

7 RELATED WORK

We divide the prior work related to our paper into three categories: (1) synthesis of functional recursive programs, (2) version-space-based synthesis, (3) bidirectional search-based synthesis. We elaborate on these categories of work. For a broader survey of program synthesis, we refer the reader to [Gulwani et al. 2017].

Synthesis of Recursive Programs. There is a large body of work on the synthesis of functional recursive programs. Various approaches have been proposed to synthesize functional recursive programs from input-output examples [Feser et al. 2015; Lubin et al. 2020; Osera and Zdancewic 2015], refinement types [Polikarpova et al. 2016], logical specifications [Itzhaky et al. 2021; Kneuss et al. 2013], and a reference implementation with desired type invariants [Farzan and Nicolet 2021]. In the following, we will mainly focus on the prior work of inductive synthesis of functional recursive programs.

Thesys [Summers 1986] and its reincarnation Igor2 [Kitzelmann and Schmid 2006] are similar to ours in the sense that they stage synthesis into (1) non-recursive program synthesis and (2) recursive program synthesis. They first synthesize non-recursive programs for the given example by a top-down search. Then, by identifying syntactic patterns, these systems "fold" the synthesized non-recursive programs into a recursive one. Similarly, Cypress [Itzhaky et al. 2021], which is for synthesizing recursive programs from separation logic specifications, also generates a satisfying straight-line program, then folds it into a generalized recursive one. Contrary to these systems, instead of exploring the space of possible foldings, which is prohibitively large in our case, we prune the search space of recursive programs by "unfolding" each candidate into a non-recursive program; we check if it can be one of the non-recursive programs synthesized earlier.

The *recursion-free approximation* in Synduce [Farzan and Nicolet 2021] is related to our block-based pruning. Synduce is a system for synthesizing a recursive program from a reference implementation and type invariants. It also synthesizes recursive programs from non-recursive programs. It eliminates recursion in a given specification by replacing each recursive call with a variable, synthesizes a satisfying non-recursive program, and then changes the variables back to their corresponding recursive calls. This method differs from ours in that we do not directly construct a recursive solution from a non-recursive one. Instead, we prune the search space of recursive programs using non-recursive ones.

Myth [Osera and Zdancewic 2015] and λ^2 [Feser et al. 2015] pioneered the idea of top-down deductive search for functional recursive programs which hypothesizes the overall structure of a program and then tries to synthesize the subcomponents. The major shortcoming of Myth is the requirement for trace-complete specifications that our system does not need. The major shortcoming of λ^2 is that it only applies deductive reasoning to a fixed set of primitive list and tree combinators such as `filter` and `map`. Our deductive reasoning is not limited to a certain set of operators but can be applied to any usable external operators thanks to the use of inverse maps.

SMyth [Lubin et al. 2020] and Burst [Miltner et al. 2022] are recently proposed systems for recursive program synthesis that do not require trace-complete specifications. SMyth explores the search space top-down and generates partial programs with holes. To alleviate the trace-completeness requirement, for each partial program, SMyth performs partial evaluation to propagate example constraints over the entire program into holes in it. However, the constraint propagation process is complex in design, thereby being costly when it comes to large programs with complex control flow. On the other hand, our tool can quickly identify infeasible candidates, significantly outperforming SMyth as already shown in Section 6.5. Burst performs bottom-up synthesis with *angelic execution* as already explained in Section 6.5. Burst inherits scalability issues of the prior bottom-up strategies where a goal-directed search in top-down strategies is missing. In contrast, our method combines top-down and bottom-up synthesis to overcome the limitation of bottom-up synthesis. In addition, Burst may run into the problem of *extensive backtracking* as explained in Section 6.5 whereas we do not have such an issue since we explore the full search space of recursive programs without any refinement process.

Eguchi et al. [Eguchi et al. 2018] have proposed a technique for synthesizing both a functional program and recursive helper functions from refinement types. Their method infers specifications of recursive helper functions by trying with a number of predefined templates. Our work focuses on synthesizing the target function when library functions are given. We expect our work can be combined with their work to synthesize the target function with a mixture of known and unknown library functions.

Version Space-Based Synthesis. To efficiently represent the set of all programs correct with respect to a given specification, the prior version space approaches to synthesis use a space-efficient data structure. FlashFill [Gulwani 2011] firstly used e-graphs like version space representations to efficiently represent the set of all correct programs and choose the best one among them.

This method is generalized in the FlashMeta [Polozov and Gulwani 2015] framework, and its instantiations [Kini and Gulwani 2015; Le and Gulwani 2014; Rolim et al. 2017] have shown successful applications of the version space approach to synthesis in various domains. These methods construct version spaces top-down as we do in our system. There have been also previous methods that construct version spaces in a bottom-up fashion. Finite tree automata (FTAs) have been used to represent version spaces of functional programs [Miltner et al. 2022; Wang et al. 2017]. In particular, Burst [Miltner et al. 2022] uses FTAs to represent the version space of *recursive* functional programs.

The major difference between our method and these previous methods is that we do not use the version space representation directly for finding a solution. Instead, we construct the version space of non-recursive programs to prune the search space of recursive programs.

DreamCoder [Ellis et al. 2021] also indirectly uses version space representations for synthesis. DreamCoder stores a large number of possible refactorings to each training program into version space representations. Those refactorings expose common sub-expressions that correspond to library functions, which DreamCoder can use for other synthesis tasks. In contrast to DreamCoder, we construct and use version spaces within a single synthesis task rather than across different tasks.

Combining Top-Down and Bottom-Up Search for Recursive Program Synthesis. The idea of combining top-down and bottom-up synthesis often appears in prior work on recursive program synthesis. λ^2 [Feser et al. 2015] enumerates open hypotheses (i.e., partial programs with holes) by a top-down deductive search and closed hypotheses by a bottom-up search. Such closed hypotheses are used to fill holes in open hypotheses. Myth [Osera and Zdancewic 2015] also enumerates expressions bottom-up up to a certain size, and uses them during a top-down deductive search.

Our work is different from these methods in that (1) we use bottom-up enumeration for collecting not only sub-expressions but also inverse maps that enable top-down propagation for arbitrary external operators, and (2) we use a combination of top-down and bottom-up synthesis not only for finding a recursive solution but also for finding all non-recursive blocks.

8 CONCLUSION

We have presented a new technique for synthesizing recursive functional programs from input-output examples. Our approach differs from prior work in that we first synthesize satisfying blocks (straight-line programs) for each input-output example, and then we prune the space of recursive programs by removing candidates that are inconsistent with the blocks. Additionally, we propose a technique we call library sampling, which accelerates deductive reasoning over a library by using sampled input-output behaviors of library functions. We have implemented our algorithm in a tool called Trio. Our comparison against the state-of-the-art synthesizers shows that Trio advances the state-of-the-art of inductive synthesis of recursive functional programs.

Data-Availability Statement. The artifact is available at Zenodo [Lee and Cho 2022].

ACKNOWLEDGMENTS

We thank the reviewers for insightful comments. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2020R1C1C1014518, 2021R1A5A1021944) and Institute for Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2021-0-00758, 2022-0-00995).

REFERENCES

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Proceedings of the 25th International Conference on Computer Aided Verification* (Saint Petersburg, Russia) *(CAV'13)*.

Shingo Eguchi, Naoki Kobayashi, and Takeshi Tsukada. 2018. Automated Synthesis of Functional Programs with Auxiliary Functions. In *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11275)*, Sukyoung Ryu (Ed.). Springer, 223–241. https://doi.org/10.1007/978-3-030-02768-1_13

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2021. DreamCoder: Bootstrapping Inductive Program Synthesis with Wake-Sleep Library Learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 835–850. https://doi.org/10.1145/3453483.3454080

Azadeh Farzan and Victor Nicolet. 2021. Counterexample-Guided Partial Bounding for Recursive Function Synthesis. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12759)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 832–855. https://doi.org/10.1007/978-3-030-81685-8_39

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI'15)*. Association for Computing Machinery, New York, NY, USA, 229–239. https://doi.org/10.1145/2737924.2737977

Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-Directed Synthesis: A Type-Theoretic Interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 802–815. https://doi.org/10.1145/2837614.2837629

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*.

Sumit Gulwani, Alex Polozov, and Rishabh Singh. 2017. *Program Synthesis*. Vol. 4. NOW. 1–119 pages. https://www.microsoft.com/en-us/research/publication/program-synthesis/

Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben N. S. Rowe, and Ilya Sergey. 2021. Cyclic Program Synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 944âĂŞ959. https://doi.org/10.1145/3453483.3454087

Dileep Kini and Sumit Gulwani. 2015. FlashNormalize: Programming by Examples for Text Normalization. In *Proceedings of the 24th International Conference on Artificial Intelligence* (Buenos Aires, Argentina) *(IJCAI'15)*. AAAI Press, 776–783.

Emanuel Kitzelmann and Ute Schmid. 2006. Inductive Synthesis of Functional Programs: An Explanation Based Generalization Approach. *Journal of Machine Learning Research* 7, 15 (2006), 429–454. http://jmlr.org/papers/v7/kitzelmann06a.html

Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis modulo Recursive Functions. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 407–426. https://doi.org/10.1145/2509136.2509555

Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*.

Woosuk Lee. 2021. Combining the top-down propagation and bottom-up enumeration for inductive program synthesis. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–28.

Woosuk Lee and Hangyeol Cho. 2022. Artifact of Inductive Synthesis of Structurally Recursive Functional Programs from Non-Recursive Expressions. https://doi.org/10.5281/zenodo.7320806

Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program sketching with live bidirectional evaluation. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–29.

Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. 2022. Bottom-up Synthesis of Recursive Functional Programs Using Angelic Execution. *Proc. ACM Program. Lang.* 6, POPL, Article 21 (jan 2022), 29 pages. https://doi.org/10.1145/3498682

Peter-Michael Osera. 2015. Program Synthesis With Types. (01 2015).

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. *ACM SIGPLAN Notices* 50, 6 (2015), 619–630.

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*

(Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 522–538. https://doi.org/10.1145/2908080.2908093

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 107–126. https://doi.org/10.1145/2814270.2814310

Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE'17)*. IEEE Press, 404–415. https://doi.org/10.1109/ICSE.2017.44

Phillip D. Summers. 1986. A Methodology for LISP Program Construction from Examples. In *Readings in Artificial Intelligence and Software Engineering*, Charles Rich and Richard C. Waters (Eds.). Morgan Kaufmann, 309–316. https://doi.org/10.1016/B978-0-934613-12-5.50028-8

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Program Synthesis Using Abstraction Refinement. *Proc. ACM Program. Lang.* 2, POPL, Article 63 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158151

Received 2022-07-07; accepted 2022-11-07