Recursive Program Synthesis

Aws Albarghouthi[1], Sumit Gulwani[2], and Zachary Kincaid[1]

[1]University of Toronto
[2]Microsoft Research

Abstract. Input-output examples are a simple and accessible way of describing program behaviour. Program synthesis from input-output examples has the potential of extending the range of computational tasks achievable by end-users who have no programming knowledge, but can articulate their desired computations by describing input-output behaviour. In this paper, we present Escher, a generic and efficient algorithm that interacts with the user via input-output examples, and synthesizes *recursive programs* implementing intended behaviour. Escher is parameterized by the *components* (instructions) that can be used in the program, thus providing a generic synthesis algorithm that can be instantiated to suit different domains. To search through the space of programs, Escher adopts a novel search strategy that utilizes special data structures for inferring conditionals and synthesizing recursive procedures. Our experimental evaluation of Escher demonstrates its ability to efficiently synthesize a wide range of programs, manipulating integers, lists, and trees. Moreover, we show that Escher outperforms a state-of-the-art SAT-based synthesis tool from the literature.

1 Introduction

Program synthesis from specifications is a foundational problem that crosses the boundaries of formal methods, software engineering, and artificial intelligence. Traditionally, specifications written in logics (such as first-order and temporal logics) have been used to synthesize programs, e.g., [15, 16]. More recently, we have witnessed renewed interest in the program synthesis question, and a shift from the traditional logical specifications to specifications presented as input-output examples, e.g., [7, 12, 10, 14, 11]. One of the main advantages of synthesis from input-output examples is that it extends the user base of synthesis techniques from algorithm and protocol designers to end-users who have no programming knowledge, but can articulate their desired computational tasks as input-output examples. For instance, recent work on synthesizing string manipulation programs from examples in spreadsheets [7] has already made the transition from research into practice, as seen in the latest release of Microsoft Excel [1]. These synthesis techniques capitalize on the fact that end-users are often interested in performing simple operations within specific domains (e.g., string manipulation), and can easily supply the synthesizer with examples demonstrating the tasks they wish to perform.

In this paper, our goal is to provide a *generic* and *efficient* synthesis algorithm that interacts with users via input-output examples, and allows for synthesis of a wide range of programs. To that end, we present ESCHER, an inductive synthesis algorithm that learns a *recursive procedure* from input-output examples provided by the user. ESCHER is parameterized by the set of *components* (instructions) that can appear in the synthesized program. Components can include basic instructions like integer addition or list concatenation, API calls, and recursive calls (i.e., instructions calling the synthesized program). This allows ESCHER to be instantiated with different sets of components and applied to different domains (e.g., integer or list manipulation). ESCHER assumes the existence of an *oracle*, simulating the user, which given an input returns an output. By interacting with the oracle, ESCHER is able to synthesize recursive programs comprised of a given set of components.

To search through the space of programs, ESCHER adopts an explicit search strategy that *alternates* between two phases: (1) *Forward Search*: in the forward search phase, ESCHER enumerates programs by picking a synthesized program and augmenting it with new components. For example, a program $f(x)$, which applies a component f to the input x, can be extended to $g(f(x))$. (2) *Conditional Inference*: in the conditional inference phase, ESCHER utilizes a novel data-structure, called a *goal graph*, which enables detecting when two programs synthesized by the forward search have to be joined by a conditional statement. For example, two programs $f(x)$ and $g(x)$ can be used to construct **if** $c(x)$ **then** $f(x)$ **else** $g(x)$.

The power of our search strategy is two-fold: (1) By alternating between forward search and conditional inference, we generate conditionals on demand, i.e., when the goal graph determines they are needed. This is in contrast to other component-based techniques, e.g., [9, 12], that use an explicit if-then-else component. (2) By adopting an explicit search strategy, as opposed to an SMT encoding like [22, 9], we do not restrict the range of synthesizable programs by the supported theories. Moreover, our explicit search strategy allows us to easily apply search heuristics, e.g., biasing the search towards synthesizing smaller programs. We have used ESCHER to synthesize a wide range of programs, manipulating integers, lists, as well as trees. Our experimental results indicate the efficiency of ESCHER and the power of our design choices, including the *goal graph* data structure for conditional inference. Furthermore, we compare ESCHER with SKETCH [22], a state-of-the-art synthesis tool, and show how ESCHER's search technique outperforms SKETCH's SAT-based technique for synthesizing programs from components.

Contributions. We summarize our contributions as follows: (1) ESCHER: a novel algorithm for synthesizing recursive programs that (a) interacts with users via input-output examples to learn programs; (b) is parameterized by the set of components allowed to appear in the program, thus providing a generic synthesis technique; and (c) uses new techniques and data structures for searching through the space of programs. (2) An implementation and an evaluation of ESCHER on a set of benchmarks that demonstrate its effectiveness at synthesizing a wide range

of recursive programs. Moreover, our results highlight the power of our goal graph data structure for conditional inference. (3) A comparison of ESCHER with a state-of-the-art synthesis tool from the literature which demonstrates ESCHER's superiority in terms of efficiency and scalability.

2 Overview

In this section, we illustrate the operation of ESCHER on a simple example. Suppose the user would like to synthesize a program that counts the number of elements in a list of integers (procedure `length` with input parameter `i`). Suppose also that ESCHER is instantiated with the following set of components: `inc`, takes an integer and returns its successor; `isEmpty`, takes a list and returns `T` (true) if the list is empty and `F` (false) otherwise; `tail`, takes a list and returns its tail; `zero`, a nullary component representing the constant `0`; and `length`, a component representing the function that we would like to synthesize. The existence of `length` as a component allows the synthesized function to be recursive, by using `length` to simulate the recursive call.

ESCHER alternates between a *forward search* phase and a *conditional inference* phase. We assume that ESCHER's alternation is guided by a heuristic function h that maps a program to a natural number, where the lower the number the more desirable the program is. For the sake of illustration, we assume that the value of h is always the size of the program, except when a program uses the same component more than once, in which case it is penalized.

Initially, the user supplies ESCHER with input-output values on which to conduct the search. Suppose the *input values* are the lists `[]`, `[2]`, and `[1,2]`. We represent input values as a *value vector* ⟨`[],[2],[1,2]`⟩. The desired *goal* value vector (outputs) corresponding to the input values is ⟨`0,1,2`⟩, where `0`, `1`, and `2` are the lengths of the lists `[]`, `[2]`, and `[1,2]`, respectively.

First Alternation. First, in the forward search phase, ESCHER creates programs that are composed of inputs or nullary components (i.e., programs of size 1). In our case, as shown in Figure 1, these are programs $\texttt{P}_1$ (the program that returns the input `i` – the identity function), and $\texttt{P}_2$ (the program that always returns `0` for all inputs). Note that each program is associated with a value vector representing its valuation, for example, the value vector of $\texttt{P}_1$ is ⟨`[],[2],[1,2]`⟩. Obviously, neither $\texttt{P}_1$ nor $\texttt{P}_2$ satisfy our goal ⟨`0,1,2`⟩. But notice that the value vector of $\texttt{P}_2$, ⟨`0,0,0`⟩, overlaps with our goal ⟨`0,1,2`⟩ in the first position, i.e., produces the correct output for the input `[]`. Therefore, the conditional inference phase determines that one way to reach our goal is to synthesize a program $\texttt{P}_r$ of the form `if` $\texttt{P}_{cond}$ `then` $\texttt{P}_2$ `else` $\texttt{P}_{else}$, where $\texttt{P}_{cond}$ is a program that evaluates to `T` on the input `[]`, and `F` on the inputs `[2]` and `[1,2]`; and $\texttt{P}_{else}$ is a program that evaluates to ⟨`1,2`⟩ on the inputs ⟨`[2],[1,2]`⟩. Intuitively, conditional inference determines that we can return `0` (program $\texttt{P}_2$) for the input `[]`, and synthesize another program to deal with inputs `[2]` and `[1,2]`.

ESCHER represents this strategy for synthesizing `length` by creating a *goal graph*, as shown in Figure 2(a). The goal graph is a novel data structure employed by ESCHER that specifies how to reach the goal ⟨`0,1,2`⟩ by synthesizing

h	Program $\mathtt{P}_i$
1	$\mathtt{P}_1 = \mathtt{i} \rightarrow \langle \mathtt{[]}, \mathtt{[2]}, \mathtt{[1,2]} \rangle$ $\mathtt{P}_2 = \mathtt{zero} \rightarrow \langle \mathtt{0,0,0} \rangle$
2	$\mathtt{P}_3 = \mathtt{tail}(\mathtt{P}_1) \rightarrow \langle \mathtt{err}, \mathtt{[]}, \mathtt{[2]} \rangle$ $\mathtt{P}_4 = \mathtt{inc}(\mathtt{P}_2) \rightarrow \langle \mathtt{1,1,1} \rangle$ $\mathtt{P}_5 = \mathtt{isEmpty}(\mathtt{P}_1) \rightarrow \langle \mathtt{T,F,F} \rangle$
3	$\mathtt{P}_6 = \mathtt{length}(\mathtt{P}_3) \rightarrow \langle \mathtt{err,0,1} \rangle$ $\mathtt{P}_7 = \mathtt{tail}(\mathtt{P}_3) \rightarrow \langle \mathtt{err,err,[]} \rangle$
4	$\mathtt{P}_8 = \mathtt{inc}(\mathtt{P}_7) \rightarrow \langle \mathtt{err,1,2} \rangle$

Fig. 1. Synthesized programs $\mathtt{P}_i$ organized by heuristic value h.

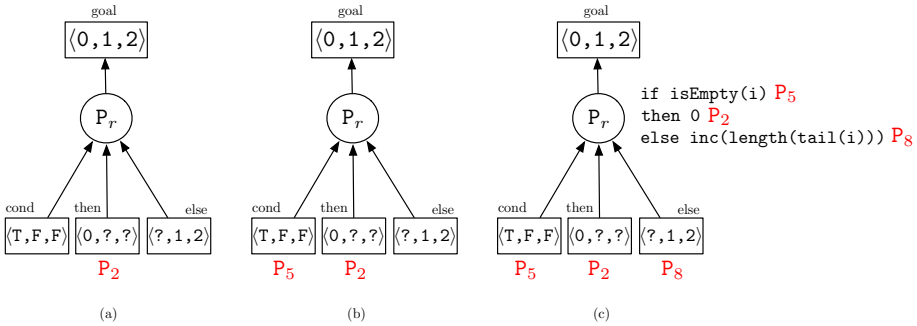

Fig. 2. Goal graph after (a) first, (b) second and third, and (c) final alternations.

a program $\mathtt{P}_r$. Specifically, we need a program $\mathtt{P}_{cond}$ that returns $\langle \mathtt{T,F,F} \rangle$ for inputs $\langle \mathtt{[]}, \mathtt{[2]}, \mathtt{[1,2]} \rangle$, and a program $\mathtt{P}_{else}$ that returns $\langle \mathtt{?,1,2} \rangle$ for inputs $\langle \mathtt{[]}, \mathtt{[2]}, \mathtt{[1,2]} \rangle$, where '?' denotes a "don't care" value, since the first input `[]` will not enter the `else` branch. Note that the `then` branch is already solvable using $\mathtt{P}_2$, and is therefore annotated with it.

Second Alternation. The forward search now synthesizes programs by applying components to programs found for the heuristic value 1. Note that ESCHER can apply `length` (the recursive call) on the input values (program $\mathtt{P}_1$), but this obviously results in a non-terminating program (namely, the program `let length(i) = length(i)`). ESCHER employs a termination argument that detects and discards non-terminating programs. Next, by applying `tail` to $\mathtt{P}_1$, we get the program $\mathtt{P}_3$ that computes $\langle \mathtt{err}, \mathtt{[]}, \mathtt{[2]} \rangle$, where `err` is a special symbol representing an error value, since `tail` is undefined on the empty list. Program $\mathtt{P}_4$ results from applying `inc` to $\mathtt{P}_2$ and generates the value vector $\langle \mathtt{1,1,1} \rangle$. Program $\mathtt{P}_5$ is generated by applying `isEmpty` to $\mathtt{P}_1$, resulting in the value vector $\langle \mathtt{T,F,F} \rangle$. Now, the conditional inference phase discovers that $\mathtt{P}_5$ solves the $\langle \mathtt{T,F,F} \rangle$ subgoal in the goal graph, and annotates it with $\mathtt{P}_5$ (see Figure 2(b)).

In the third alternation, new programs are generated, but none of our goals are solved.

Final Alternation. Finally, forward search applies `inc` to $\mathtt{P}_6$ and generates $\mathtt{P}_8$ with the value vector $\langle \mathtt{err,1,2} \rangle$. Conditional inference recognizes that this

program solves the subgoal $\langle$`?,1,2`$\rangle$. Since all subgoals of our main goal $\langle$`0,1,2`$\rangle$ are now solved, we can synthesize the final program `if` P_5 `then` P_2 `else` P_8. Figure 2(c) shows the result produced by ESCHER, which satisfies the given input-output values, and extrapolates to the behaviour intended by the user.

It is important to note that each program is associated with a value vector representing its execution on the given inputs. This allows us to restrict the search space by treating programs with equivalent value vectors as equivalent programs. This *observational equivalence* property can reduce the search space drastically, as will be demonstrated experimentally in Section 4. Moreover, our use of goal graphs allows us to efficiently synthesize programs that contain conditionals. Other component-based techniques like [9, 12] approach this problem by using an if-then-else component; in Section 4, we demonstrate the advantages of the goal graph over this approach experimentally.

3 The Escher algorithm

In this section, we provide the basic definitions required for the rest of the paper, present the ESCHER algorithm and discuss its properties and practical considerations.

3.1 Definitions

Synthesis Task. We define a *synthesis task* $\mathcal{S}$ as a pair $(Ex, Comps)$, where Ex is a list of input-output examples, and $Comps$ is the set of components allowed to appear in the synthesized program. We assume that $Comps$ contains a special component self which is treated as a recursive call to the synthesized program. Every component c is associated with an arity $\mathsf{a}(c) \in [0, \infty)$, indicating the number of input parameters c accepts. We define $\mathsf{a}(\mathsf{self})$ to be the number of input parameters in the examples in Ex. We assume, without loss of generality, that each component returns one value.

We use $\mathcal{V}$ to denote the set of *values* (which may include integers, Booleans, lists of integers, etc.). We make three assumptions about $\mathcal{V}$: equality must be decidable on $\mathcal{V}$, there is a well-founded relation $\prec \subseteq \mathcal{V} \times \mathcal{V}$, and $\mathcal{V}$ must contain a distinguished err value that denotes the result of an erroneous computation (e.g., a run-time exception, a type error, or a non-terminating computation), as well as the Boolean values `T` and `F`. We assume the existence of a total function eval that takes a component c and an $\mathsf{a}(c)$-tuple of values $I \in \mathcal{V}^{\mathsf{a}(c)}$, and returns a value representing the result of applying c to I. For example, if c is integer division, then $\mathsf{eval}(c, (6, 3)) = 2$ and $\mathsf{eval}(c, (4, 0)) = \mathsf{err}$. For recursive program synthesis, we treat the evaluation of the target component self as a call to the *Oracle* (that is, self is evaluated by the user). We memoize such calls to avoid making repeated queries to the *Oracle* for the same input.

Programs. Given a synthesis task $(Ex, Comps)$, we define a *program* P over $Comps$ using the following grammar:

$$
\begin{array}{rcll}
P \in Program & ::= & \texttt{if } P_{cond} \texttt{ then } P_{then} \texttt{ else } P_{else} & \\
& | & c(P_1, \ldots, P_{\mathsf{a}(c)}) & \text{if } \mathsf{a}(c) > 0 \\
& | & c & \text{if } \mathsf{a}(c) = 0 \\
& | & x_j & 1 \leq j \leq \mathsf{a}(\mathsf{self})
\end{array}
$$

where x_j is an input parameter, and c is a component. We use *Vars* to denote the set of all input parameters.

Note that our programming language is untyped (programs encountering a run-time type error evaluate to err), first order (although the components themselves may be higher-order), and purely functional.

3.2 Algorithm Description

Problem Definition. We start by formalizing the synthesis problem as follows. We use (in_i, out_i) to denote the ith input-output example in Ex (where $in_i, out_i \in \mathcal{V}$), $v[j]$ to denote the jth element of a value vector v, and define a function $\mathsf{eval}_v : Program \to \mathcal{V}^{|Ex|}$ as follows:

$$\mathsf{eval}_v(x_j)[i] = in_i[j]$$
$$\mathsf{eval}_v(c)[i] = \mathsf{eval}(c, ())$$
$$\mathsf{eval}_v(c(P_1, \ldots, P_n))[i] = \mathsf{eval}(c, (\mathsf{eval}_v(P_1)[i], \ldots, \mathsf{eval}_v(P_n)[i]))$$
$$\mathsf{eval}_v(\texttt{if } P_{cond} \texttt{ then } P_{then} \texttt{ else } P_{else})[i] = \begin{cases} \mathsf{eval}_v(P_{then})[i] & \text{if } \mathsf{eval}_v(P_{cond})[i] = \texttt{T} \\ \mathsf{eval}_v(P_{else})[i] & \text{if } \mathsf{eval}_v(P_{cond})[i] = \texttt{F} \\ \mathsf{err} & \text{otherwise} \end{cases}$$

A *(sub)goal* is a vector whose values range over program values $\mathcal{V}$ and a distinguished "don't care" value `?`. The *root goal* is a goal consisting of the desired outputs from the given examples, namely $root = \langle out_1, \cdots, out_{|Ex|} \rangle$. For a value vector v and a (sub)goal g, we say that v *matches* g (and write $match(v, g)$) if for every i, either $v[i] = g[i]$ or $g[i] = \texttt{?}$.

The program synthesis problem can be formalized as follows: find a program P such that $\mathsf{eval}_v(P)$ matches $root$.

Escher Formalized. The synthesis procedure of Escher is pictured in Figure 3 as a nondeterministic transition system. Escher takes as input a synthesis task $(Ex, Comps)$ and synthesizes a program that matches the input-output examples given in Ex. In the following, we first give a high-level overview of Escher, and then describe the system more formally.

A configuration of Escher is a triple $\langle \mathsf{syn}, \mathsf{goalGraph}, \mathsf{ex} \rangle$ consisting of a set of synthesized programs syn, a goal graph $\mathsf{goalGraph}$, and a list of input-output examples ex. The procedure begins by applying the Init rule, which initializes syn to be the set of input variables *Vars* and $\mathsf{goalGraph}$ to be the goal graph consisting only of a single goal node *root* (representing the desired outputs obtained from the examples Ex) and initializes ex to be Ex.

The rule Forward implements the forward search part of Escher: a new program is added to syn by applying a component to a vector of programs that have already been synthesized (members of syn). The rules SplitGoal and Resolve implement the conditional inference part of the search by manipulating the goal graph – we will explain these rules further in the following. The Saturate rule adds new input-output examples to ex; if Escher synthesizes a recursive program, we must check that the program also produces the correct

$$\frac{}{\langle Vars, (\{\{root\}, \emptyset, \emptyset, root), Ex\rangle} \textsc{Init}$$

$$\frac{c \in Comps \qquad P_1 \in \mathsf{syn} \qquad \cdots \qquad P_{\mathsf{a}(c)} \in \mathsf{syn} \qquad P = c(P_1, \ldots, P_{\mathsf{a}(c)})}{\langle \mathsf{syn}, \mathsf{goalGraph}, \mathsf{ex}\rangle \rightarrow \langle \mathsf{syn} \cup \{P\}, \mathsf{goalGraph}, \mathsf{ex}\rangle} \textsc{Forward}$$

$$\frac{\begin{array}{c} cond \in \mathbb{B}^{|\mathsf{ex}|} \qquad g \in G \qquad r \text{ is fresh} \\ bthen = g|cond \qquad belse = g|\neg cond \\ G' = G \cup \{cond, bthen, belse\} \qquad R' = R \cup \{r\} \\ E' = E \cup \{(r, g), (cond, r), (bthen, r), (belse, r)\} \end{array}}{\langle \mathsf{syn}, (G, R, E, root), \mathsf{ex}\rangle \rightarrow \langle \mathsf{syn}, (G', R', E', root), \mathsf{ex}\rangle} \textsc{SplitGoal}$$

$$\frac{\begin{array}{c} P_1, P_2, P_3 \in \mathsf{syn} \qquad r \in R \qquad (r, g_1), (r, g_2), (r, g_3) \in E \\ match(\mathsf{eval}_v(P_1), g_1) \qquad match(\mathsf{eval}_v(P_2), g_2) \qquad match(\mathsf{eval}_v(P_3), g_3) \\ P = \texttt{if } P_1 \texttt{ then } P_2 \texttt{ else } P_3 \end{array}}{\langle \mathsf{syn}, (G, R, E, root), \mathsf{ex}\rangle \rightarrow \langle \mathsf{syn} \cup \{P\}, (G, R, E, root), \mathsf{ex}\rangle} \textsc{Resolve}$$

$$\frac{P \in \mathsf{syn} \qquad match(\mathsf{eval}_v(P), root) \qquad \mathsf{ex}{\downarrow_P} \not\subseteq \mathsf{ex}}{\langle \mathsf{syn}, (G, R, E, root), \mathsf{ex}\rangle \rightarrow \langle \mathsf{syn}, (\{root'\}, \emptyset, \emptyset, root'), \mathsf{ex} \cup \mathsf{ex}{\downarrow_P}\rangle} \textsc{Saturate}$$

$$\frac{P \in \mathsf{syn} \qquad match(\mathsf{eval}_v(P), root) \qquad \mathsf{ex}{\downarrow_P} \subseteq \mathsf{ex}}{\langle \mathsf{syn}, (G, R, E, root), \mathsf{ex}\rangle \rightarrow P} \textsc{Terminate}$$

Fig. 3. Escher synthesis algorithm

results for all the recursive calls in order to ensure that the synthesized program is a solution to the synthesis task. Finally, the Terminate rule terminates the algorithm when a program matching the root goal has been synthesized.

The Goal Graph We now describe the data structure and technique used for synthesizing conditionals. A goal graph is a bipartite graph $(G, R, E, root)$, where:

- G is a set of value vectors representing *goals* that need to be achieved,
- R is a set of *resolvers* connecting goals to subgoals,
- $E \subseteq G \times R \cup R \times G$ is a set of edges connecting goals and resolvers, and
- $root \in G$ is a distinguished *root goal*.

We assume that each resolver $r \in R$ has a single outgoing edge $(r, g) \in E$, the target of which is called the *parent* of r, and three incoming edges $(g_1, r), (g_2, r), (g_3, r)$, denoting the *cond*, *then*, and *else* goals that need to be synthesized to synthesize a program solving goal g. We call g_1, g_2, and g_3, *subgoals* of r and g. We also assume that, with the exception of the root goal *root* (which has no outgoing edges), every goal has at least one outgoing edge (i.e., it is a subgoal of at least one resolver).

Example 1. Consider the goal graph in Figure 2. The set of goals $G = \{\langle \texttt{0,1,2}\rangle, \langle \texttt{T,F,F}\rangle, \langle \texttt{0,?,?}\rangle, \langle \texttt{?,1,2}\rangle\}$, the set of resolvers $R = \{\texttt{P}_r\}$, and root

of the graph is $\langle \texttt{0,1,2} \rangle$. The graph specifies that in order to synthesize a program for $\langle \texttt{0,1,2} \rangle$, one could synthesize three programs satisfying the three other vectors in G. □

At a high-level, a goal graph $\mathsf{goalGraph}$ can be viewed as an AND-OR graph. That is, the goal graph specifies that to synthesize a program satisfying a goal g, then programs satisfying *all* subgoals g_1, g_2, g_3 of *one* of the resolvers r (s.t. $(r, g) \in E$) have to be synthesized.

The rule SplitGoal updates a goal graph by including a new resolver for a given goal. This is accomplished by selecting an arbitrary Boolean vector $cond \in \mathbb{B}^{|\mathsf{ex}|}$ and a goal g. From $cond$ and g, we compute a pair of *residual goals* $bthen = g|cond$ and $belse = g|\neg cond$, which agree with g on positions where $cond$ is true (or false, in the case of $belse$), and otherwise have don't-care values. Formally,

$$(g|cond)[i] = \begin{cases} g[i] & \text{if } cond[i] \\ \texttt{?} & \text{otherwise} \end{cases} \qquad (g|\neg cond)[i] = \begin{cases} g[i] & \text{if } \neg cond[i] \\ \texttt{?} & \text{otherwise} \end{cases}$$

SplitGoal creates a new resolver for g with three new sub-goals: one for $cond$, one for $bthen$, and one for $belse$, and adds them to the goal graph.

Termination Argument. The procedure described in the preceding is suitable for synthesizing non-recursive programs, but a termination argument is required to synthesize recursive functions. To see why a termination argument is required, consider that (in its absence) the program $\mathsf{self}(x_1, \ldots, x_{\mathsf{a(self)}})$ is always a solution (since this program always matches the *root* goal). This solution should be excluded from the search space because it does not terminate.

We remove non-terminating programs from the search space by redefining the eval_v on the recursive component self so that an error is produced on arguments that are not decreasing, according to the well-founded relation $\prec$ on $\mathcal{V}$. Formally, we define

$$\mathsf{eval}_v(\mathsf{self}(P_1, \ldots, P_{\mathsf{a(self)}}))[i] = \begin{cases} \mathsf{eval}(\mathsf{self}, arg) & \text{if } arg \prec^* in_i \\ \mathsf{err} & \text{otherwise} \end{cases}$$

where $arg = (\mathsf{eval}_v(P_1)[i], \ldots, \mathsf{eval}_v(P_{\mathsf{a(self)}})[i])$ and $\prec^*$ indicates the well-founded relation $\prec$ on $\mathcal{V}$ extended to the lexicographic well-founded relation on $\mathcal{V}^{\mathsf{a(self)}}$.

Example 2. Recall the example from Section 2. To ensure termination of the resulting program for computing list length, Escher enforced that the list with which the recursive call to `length` is made follows the common well-founded order for the list data type: the length of the list is decreasing. For example, suppose we synthesize the program `length(i)` (where `i` is the only input variable) using Forward. Then we may not apply the Terminate rule, since $\mathsf{eval}_v(\texttt{length(i)}) = \langle \mathsf{err,err,err} \rangle$, which does not match the root goal $\langle \texttt{0,1,2} \rangle$.

Saturation. Finally, we discuss our Saturate rule. The reason for including this rule is illustrated by the following example:

Example 3. Consider the `length` synthesis task introduced in Section 2, and suppose that our examples are (`[]`,`0`) and (`[1,2]`,`2`), such that *root* is $\langle 0, 2\rangle$. Let P be the program

```
if isEmpty(i) then 0
else if isEmpty(tail(i)) then 0
     else inc(length(tail(i)))
```

Then $match(P, root)$, but P is not a solution to the synthesis task. □

The problem with the above example is that eval_v uses the *Oracle* to evaluate recursive calls rather than the synthesized program. This is required because Escher constructs programs in a bottom-up fashion, so eval_v must be able to evaluate self before a candidate solution is fully constructed. So, on the above example, P returns `1` result on input `[1,2]`, but $match(P, root)$ holds because it uses the *Oracle* to evaluate `length(tail(i))`. The Saturate rule (and the $\mathsf{ex}{\downarrow_P}{\subseteq}\ \mathsf{ex}$ side-condition of Terminate) resolves this problem, as we will describe in the following.

We define $\mathsf{ex}{\downarrow_P}$ to be the set of all input-output examples (I, O) such that there is some recursive call in P that evaluates $\mathsf{self}(I)$ for one of the input-output examples in ex. On our example, $\mathsf{ex}{\downarrow_P} = \{(\texttt{[2]},\texttt{1})\}$. So Saturate adds this new example to ex. For completeness, we formally define $\mathsf{ex}{\downarrow_P}$ in the Appendix.

Intuitively, the $\mathsf{ex}{\downarrow_P}$ is the set of input-output examples upon which the examples in ex depend. A *saturated* set of examples (one in which $\mathsf{ex}{\downarrow_P}{\subseteq}\ \mathsf{ex}$) does not depend on anything – so if a program P is correct on a saturated example set, then it is guaranteed to be correct for all the examples. The Saturate rule simply adds such "dependent examples" to the set of examples for which we are obligated to prove correctness.

Completeness We conclude this section with a statement of the completeness of our synthesis algorithm. Escher is complete in the sense that if there exists a solution to the synthesis task within the search space, it will eventually find one. Theorem 1 states this property formally.

Theorem 1 (Relative Completeness). *Given a synthesis task* $(Ex, Comps)$*, suppose there exists a program* $P \in Program$ *such that* $match(\mathsf{eval}_v(P), goal)$*,*[1] *where* $goal = \langle out_1, \ldots, out_{|Ex|}\rangle$ *Then for all reachable configurations* $\langle \mathsf{syn}, \mathsf{goalGraph}, \mathsf{ex}\rangle$ *of* Escher*, there exists a run of the algorithm ending in a solution to the synthesis problem.*

Assuming a natural fairness condition on sequences of Escher rule applications, we have an even stronger result: if a solution to the synthesis task exists, Escher will find one.

[1] Note that this condition implies that P terminates according to the argument above.

3.3 Search guidance

In this section, we discuss techniques we use to guide the search procedure used in ESCHER. These techniques are essential for turning the naïve transition system presented in Figure 3 into a practical synthesis algorithm.

Heuristic search. In a practical implementation of ESCHER, we require a method for choosing which rule to apply in any given configuration. In particular, FORWARD has a large branching factor, so it is necessary to determine which component to apply to which subprograms at every step.

Our method is based on using a *heuristic function* $h : Program \to \mathbb{N}$ that maps programs to natural numbers. The lower the heuristic value of a program, the more it is desired.

A simple example of a heuristic function is the function mapping each program to its size. When ESCHER is instantiated with this heuristic function, the search is biased towards smaller (more desirable) programs. The design of more sophisticated heuristic functions for program synthesis is an important and interesting problem, but is out of the scope of this paper. A promising approach based on machine learning is presented in [17].

Observational Equivalence Reduction. The synthesis problem solved by ESCHER requires a program to be synthesized that matches a given list of input-output examples. Programs that evaluate to the same outputs for the inputs given in ex are indistinguishable from the perspective of this task. This idea yields a technique for reducing the size of search space, which we call observational equivalence reduction.

We define an equivalence relation $\equiv$ on programs such that $P \equiv Q$ iff $\mathsf{eval}_v(P) = \mathsf{eval}_v(Q)$. Whenever a new program P is synthesized, ESCHER checks whether a program Q has already been synthesized such that $P \equiv Q$: if such a program Q exists, the program P is discarded. This ensures that at most one representative from each equivalence class of $\equiv$ is synthesized. The correctness of observational equivalence reduction is implied by the following proposition.

Proposition 1. *Let P, Q, Q' be programs such that $Q \equiv Q'$, and that P is a solution to the synthesis task (i.e., $match(\mathsf{eval}_v(P), root)$). Let P' be the program obtained from P by replacing every instance of Q with Q'. Then P' is also a solution to the synthesis task.*

A corollary of this proposition is that our completeness theorem (Theorem 1) still holds in the presence of observational equivalence reduction.

Rule scheduling. In this section, we briefly comment on some practical considerations involved in scheduling the rules presented in Figure 3.

In our implementation of ESCHER, the FORWARD rule is applied in a dynamic programming fashion, as demonstrated in Section 2. Whenever a new program P is synthesized, we apply SATURATE/TERMINATE to check if P is a solution to the synthesis problem. If not, we apply RESOLVE eagerly to close as many goals as possible. We then apply SPLITGOAL if P matches *some* positions of an open goal. For example, if $\langle \texttt{0,1,2,3} \rangle$ is an open goal and P computes $\langle \texttt{0,0,2,2} \rangle$, then we apply SPLITGOAL with the Boolean condition $\langle \texttt{T,F,T,F} \rangle$ (i.e., for each

```
let hbal_tree n =
  if leq0(div2(n)) then createLeaf(0)
  else createNode(0, hbal_tree(div2(dec(n))), hbal_tree(div2(n)))

let stutter l =
  if isEmpty(x) then emptyList
  else cons(head(x), cons(head(x), stutter(tail(x))))
```

Fig. 4. Output by ESCHER for height balanced binary tree `hbal_tree`, assuming `n > 0`, and `stutter`.

position i, the condition at position i is `T` if P matches the goal at i and `F` otherwise).

Since the SATURATE burdens the user by requiring them to provide additional input/output examples, it may be desirable to schedule rules so that SATURATE is rarely applied. To accomplish this goal, we may assign high heuristic values to programs which require additional user input to bias the search away from applying the SATURATE that SATURATE.

4 Implementation and Evaluation

We have implemented an OCaml prototype of ESCHER in a modular fashion, allowing heuristic functions and components written as OCaml functions to easily be plugged in. Our goal in evaluating ESCHER is as follows: (1) Study the effectiveness of ESCHER on a broad range of problems requiring recursive solutions. (2) Evaluate the performance impact of ESCHER's key goal graph concept and its observational equivalence search guidance heuristic. (3) Evaluate ESCHER against SAT/SMT-based techniques by comparing it to the state-of-the-art tool SKETCH [22].

Benchmarks. Our benchmark suite consists of a number of recursive integer, list, and tree manipulating programs, which were drawn from functional programming assignments, standard list and tree manipulation examples, and classic recursive programming examples. The types of programs we have synthesized with ESCHER include tail recursive, divide-and-conquer, as well as mutually recursive programs, thus demonstrating the flexibility and power of the algorithm in this setting. For example, Figure 4 shows two functions synthesized by ESCHER. The first one, `hbal_tree`, constructs a height-balanced binary tree of a given size `n` using a divide-and-conquer recursive strategy to construct the left and right subtrees of each node in the tree separately. The second function, `stutter`, duplicates each element in a list. Due to lack of space, we describe our benchmark set in detail in the Appendix.

In order to synthesize these programs, we supplied ESCHER with a base set of components shown in Figure 5. For all synthesis tasks, this same set of components was used. Our goal with this decision is two-fold: (1) Model a user-friendly environment where the user is not forced to provide a different focused set of components for different tasks, since this requires non-trivial thinking on

Component Type	Supplied Components
Boolean	`and, or, not, equal, leq0, isEmpty`
Integer	`plus, minus, inc, dec, zero, div2`
List	`tail, head, cat, cons, emptyList`
Tree	`isLeaf, treeVal, treeLeft, treeRight createNode, createLeaf`

Fig. 5. Base set of components used in experiments.

#	Program	EscherS	ObsEqOff	GGOff	AllOff	EscherD
	Tree Programs					
1	`collect_leaves`	0.016	0.052	–	–	2.120
2	`count_leaves`	0.008	0.024	3.212	4.424	2.680
3	`hbal_tree`	0.244	–	–	–	–
4	`nodes_at_level`	4.040	–	–	–	–
	List Programs					
5	`compress`	1.000	–	–	–	–
6	`concat`	0.020	0.016	0.772	1.240	0.076
7	`drop`	0.004	0.012	0.280	0.332	0.004
8	`insert`	0.976	–	–	–	–
9	`last`	0.000	0.028	0.096	0.148	0.096
10	`length`	0.004	0.016	0.040	0.044	0.120
11	`reverse`	0.064	5.864	3.656	5.440	0.112
12	`stutter`	0.160	43.515	14.069	20.809	–
13	`sum`	0.008	0.164	0.372	0.548	0.092
14	`take`	0.084	9.233	14.489	18.561	2.776
	Integer Programs					
15	`fib`	0.068	3.092	–	–	–
16	`gcd`	0.004	0.008	–	–	0.004
17-1	`iseven`	0.008	0.008	0.004	0.008	1.096
17-2	`isodd`	0.012	0.016	0.008	0.012	1.100
18	`modulo`	0.016	0.304	0.204	0.272	0.024
19	`mult`	0.040	4.748	5.756	7.564	–
20	`square`	0.068	4.680	8.345	13.685	1.940
21	`sum_under`	0.004	0.024	1.220	2.044	1.168

Fig. 6. Synthesis time of Escher instantiations using the 22 components indicated in Figure 5. Time is in seconds. '–' denotes a timeout, where the time limit is 60 seconds.

the part of the user. (2) Demonstrate Escher's ability to synthesize non-trivial programs in the presence of superfluous components.

Our base components cover most basic Boolean, integer, list, and tree operations. For example, Boolean components supply all logical connectives as well as equality checking.

Experimental Setup and Results. We will use EscherS to refer to an instantiation of Escher with the size heuristic for search guidance, and EscherD to refer to an instantiation with the program depth heuristic (i.e., $h(P)$ is program depth). To study the effects of the goal graph, we implemented a configuration of Escher called GGOff that synthesizes conditionals using the technique employed by [9, 12], where an if-then-else compo-

#Comps	`mult`	`modulo`	`sum_under`	`iseven`	`isodd`
0	0.705	0.025	24.339	0.011	0.012
1	12.001	0.069	36.810	0.016	0.015
2	12.570	0.081	42.909	0.018	0.021
3	16.703	0.119	40.952	0.017	0.025
4	16.681	0.188	59.905	0.017	0.028
5	36.269	0.129	66.622	0.020	0.026

Fig. 7. Sketch evaluation.

nent is used to synthesize conditionals and the goal graph is disabled. To study the effects of observational equivalence, we implemented a configuration of Escher called ObsEqOff, where observational equivalence is not checked and all programs are considered. Additionally, AllOff represents a configuration of Escher where both observational equivalence and the goal graph are not used. Except for EscherD, all aforementioned configurations use the size of the program heuristic to guide the search.

We started all configurations of Escher with a minimal number of input-output tuples required to synthesize a correct program for each benchmark (i.e., without having to ask the *Oracle*). Figure 6 shows the number of seconds required by each configuration of Escher to synthesize a correct implementation of the given synthesis task. For example, row 4 shows that EscherS synthesizes the tree manipulating program `nodes_at_level` in 4 seconds, whereas ObsEqOff,GGOff, AllOff, as well as EscherD fail to produce a result in the allotted time.

Our results demonstrate the ability of Escher to synthesize non-trivial programs in a very small amount of time, typically less than a second. Moreover, for a large number of programs, not using the goal graph causes the tool to timeout (e.g., `gcd`) or spend a considerable amount of time in synthesis (e.g., `hbal_tree`). This demonstrates the power of our technique for synthesizing conditionals in comparison with the naïve method of using an if-then-else component. A similar effect is observed in ObsEqOff, where all synthesized programs are considered. We also observe that on our suite of benchmarks, the program size heuristic outperforms the depth heuristic.

Comparison with Sketch. Sketch [22] is a state-of-the-art synthesis tool that accepts as input a *sketch* (partial program) and a reference implementation (oracle). The sketch is written in a C-like programming language that includes the construct "`??`", denoting an unknown constant value. Sketch then uses SAT-solving to find constants to replace each occurrence of `??` with an integer such that the resulting program is equivalent to a reference implementation. In [9], the authors compare their SMT component-based synthesis technique for straight line programs against Sketch. This is done by encoding the task for searching for a straight line program composed of a set of components as a sketch. For example, given a choice of two components, one can encode the choice as `if (??) then comp1() else comp2()`

To evaluate Escher's heuristic search approach against SAT-based techniques, we encoded our component-based synthesis tasks as sketches in a similar fashion to [9], with the addition that a reference specification was also encoded as a component (in order to simulate a recursive call). To ensure termination of the synthesized program, we encoded the same termination argument used by Escher (Section 3). Our encoding produces sketches of size linear in the number of components.

There are two limitations to applying Sketch to our suite of benchmarks: (1) Sketch can only be applied to the integer benchmarks, since it does not support tree and list data structures; and (2) in order to successfully synthesize

programs with SKETCH, we had to supply it with the top level conditional in each benchmark, thus aiding SKETCH by restricting the search space.

On `fib`, `gcd`, and `sum_under`, SKETCH exhausted the allotted 2GB of memory. On `square`, SKETCH returned an error. Figure 7 shows the time taken by SKETCH[2] on the rest of the integer benchmarks where it successfully synthesized a program. The column #COMPS denotes the number of superfluous components provided in the sketch (i.e., components not required for synthesizing the task in question). For example, for `mult`, when the number of components is exactly what is required for synthesis, SKETCH generated a program in 0.71 seconds, but when the number of extra components is 3, SKETCH required 16.7 seconds to synthesize a solution. We observe that the time taken by SKETCH steadily increases in `mult` and `sum_under` as we increase the number of superfluous components. In contrast, ESCHER's results in Figure 6 were obtained by supplying ESCHER with all the 22 components, demonstrating the scalability of ESCHER in the presence of superfluous components.

In summary, our results demonstrate the efficiency of ESCHER at synthesizing a broad range of programs, and emphasize the power of our goal graph data structure at synthesizing conditionals. Moreover, we show that our prototype implementation of ESCHER can outperform a state-of-the-art SAT-based synthesis tool at synthesizing recursive programs from components.

5 Related Work

For a recent survey of various techniques and interaction models for synthesis from examples, we refer the reader to [8].

In version-space algebras, the idea is to design data structures that succinctly represent all expressions/programs that are consistent with a given set of examples. Mitchell [18] pioneered this technique for learning Boolean functions. Lau et al. [14] adapted the concept to *Programming By Demonstration* (PBD) [2], where the synthesizer learns complex functions for text editing tasks. More recently, version-space algebras have been used for data manipulation in spreadsheets, e.g., string transformations [7], number transformations [21], and table transformations [11]. These techniques are limited to domain-specific languages, and different synthesis algorithms are required for different domains. In contrast, ESCHER is parameterized by the components used, thus offering a flexible domain-agnostic synthesis solution.

Explicit search techniques enumerate the space of programs until a program satisfying the given examples is found. This appears in the context of AI planning [19, 4], where the search is directed by a goal-distance heuristic. Machine learning techniques have also been used for guiding the search using textual features of examples [3]. These techniques have been mostly limited to synthesis of straight line programs. In contrast, ESCHER's search technique can discover recursive programs.

SAT and SMT solvers have also been used for synthesis. Sketching [22] is the most prominent technique in this category. It accepts a program with holes, and

[2] Only synthesis time is reported – verification time is not counted.

uses a SAT solver to fill the holes with constants to satisfy a given specification (represented as a program). This is performed by bit-blasting the program and encoding it as a formula. In [12], SMT solvers are used to synthesize straight line bit-manipulating programs by interacting with the user via input-output examples. In contrast to these techniques, ESCHER is not restricted by the theories supported by the SMT solver. Also, as we have shown experimentally, sketching is highly sensitive to superfluous components, and even required the top-level conditional in the program to be supplied for successful synthesis. Moreover, ESCHER's heuristic search strategy provides a direct way of adding search preferences/guidance, without requiring a complex encoding of preferences as SMT formulas.

The field of inductive logic programming (ILP) was spawned by the work of Shapiro on the Model Inference System (MIS) [20] and by the work of Summers on LISP synthesis [23], among others. Flener and Yilmaz [6] present a nice survey of this rich area. MIS performs synthesis in an interactive manner using search (like ESCHER). However, it scales by fixing errors in a current (incorrect) program. We do not fix incorrect programs but build one from scratch. The idea in Summer's work and its recent incarnation [13] is to start by synthesizing a non-recursive program for the given examples. Then, by looking for syntactic patterns in the synthesized program, the non-recursive program is generalized into a recursive one. ESCHER's approach differs significantly from these techniques, since the whole synthesis algorithm is based on search, and there is no distinction between finding non-recursive programs and generalization. Moreover, ESCHER does not require a "good" set of examples to successfully synthesize a program. Instead, ESCHER can interactively query the user/oracle for more examples (if the initial set does not suffice) until it finds a solution. Summer's line of work was also extended by Flenner in his DIALOGS system [5], which is also interactive and features abduction as well (like ESCHER, which abduces conditions of if-then-else statements). However, ESCHER is based on heuristic search to make the process efficient, while DIALOGS uses a non-deterministic algorithm in order to also synthesize alternative programs.

6 Conclusion and Future Work

We have presented ESCHER, a generic and efficient algorithm that interacts with the user via input-output examples, and synthesizes recursive programs implementing intended behaviour. Our work presents a number of interesting questions for future consideration. On the technical side, we would like to extend ESCHER to synthesize loops, alongside recursion. To improve ESCHER's ability to synthesize constants, it would be interesting to combine ESCHER's heuristic search with an SMT-based search. For example, ESCHER can heuristically decide to use an SMT solver to check if there is a solution that uses synthesized constants within n steps for a given input-output example. On the application side, it would be interesting to study the applicability of ESCHER as an *intelligent tutoring system*, where students can learn recursion as a programming paradigm by interacting

with the synthesizer, e.g., for suggesting different solutions or providing hints for completing student solutions.

References

1. Flash Fill (Microsoft Excel 2013 feature). `http://research.microsoft.com/users/sumitg/flashfill.html`.
2. *Your wish is my command: programming by example*. Morgan Kaufmann Publishers Inc., 2001.
3. S. G. B. L. Aditya Menon, Omer Tamuz and A. Kalai. A machine learning framework for programming by example. In *ICML'13*, 2013.
4. B. Bonet and H. Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001.
5. P. Flener. Inductive logic program synthesis with dialogs. In *Inductive Logic Programming Workshop*, pages 175–198, 1996.
6. P. Flener and S. Yilmaz. Inductive synthesis of recursive logic programs: Achievements and prospects. *J. Log. Program.*, 41(2-3):141–195, 1999.
7. S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proc. of POPL'11*, pages 317–330.
8. S. Gulwani. Synthesis from examples: Interaction models and algorithms. *Proc. of SYNASC'12*. Invited talk paper.
9. S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *Proc. of PLDI'11*, pages 62–73.
10. S. Gulwani, V. A. Korthikanti, and A. Tiwari. Synthesizing geometry constructions. In *Proc. of PLDI'11*, pages 50–61.
11. W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *Proc. of PLDI'11*, pages 317–328.
12. S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proc. of ICSE'10*, pages 215–224.
13. E. Kitzelmann and U. Schmid. Inductive synthesis of functional programs: An explanation based generalization approach. *JMLR*, 7:429–454, 2006.
14. T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld. Programming by demonstration using version space algebra. *JMLR*, 53(1-2):111–156, Oct. 2003.
15. Z. Manna and R. J. Waldinger. A deductive approach to program synthesis. *ACM TOPLAS*, 2(1):90–121, 1980.
16. Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 6(1):68–93, Jan. 1984.
17. A. Menon, O. Tamuz, S. Gulwani, B. Lampson, and A. Kalai. A machine learning framework for programming by example. In *ICML*, 2013. To appear.
18. T. M. Mitchell. Generalization as search. *Artif. Intell.*, 18(2):203–226, 1982.
19. D. Nau, M. Ghallab, and P. Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.
20. E. Y. Shapiro. *Algorithmic Program DeBugging*. MIT Press, Cambridge, MA, USA, 1983.
21. R. Singh and S. Gulwani. Synthesizing number transformations from input-output examples. In *Proc. of CAV'12*, pages 634–651.
22. A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *Proc. of ASPLOS'06*, pages 404–415.
23. P. D. Summers. A methodology for lisp program construction from examples. *J. ACM*, 24(1):161–175, Jan. 1977.

A Saturation

We now define $\mathsf{ex}{\downarrow_P}$, from Section 3, formally:

$$\mathsf{ex}{\downarrow_P} = \{(I, Oracle(I)) : \mathsf{self}(P_1, \ldots, P_{\mathsf{a(self)}}) \text{ in } P, i \in [1, |\mathsf{ex}|], I = \langle \mathsf{eval}_v(P_1)[i], \ldots, \mathsf{eval}_v(P_{\mathsf{a(self)}})[i] \rangle\}$$

$$\mathsf{ex}{\downarrow_P} = \begin{cases} \mathsf{ex}{\downarrow_{P_1}} \cup \cdots \cup \mathsf{ex}{\downarrow_{P_{\mathsf{a(self)}}}} \cup \overrightarrow{\mathsf{eval}_v}(P_1, \ldots, P_{\mathsf{a(self)}})I & \text{if } P = \mathsf{self}(P_1, \cdots, P_{\mathsf{a(self)}}) \\ \mathsf{ex}{\downarrow_{P_1}} \cup \cdots \cup \mathsf{ex}{\downarrow_{P_n}} & \text{if } P = c(P_1, \cdots, P_n) \\ \mathsf{ex}{\downarrow_{P_{cond}}} \cup \mathsf{exT}{\downarrow_{P_{then}}} \cup \mathsf{exF}{\downarrow_{P_{else}}} & \text{if } P = \textbf{if } P_{cond} \textbf{ then } P_{then} \textbf{ else } P_{else} \end{cases}$$

where

$$\overrightarrow{\mathsf{eval}_v}(P_1, \cdots, P_n) = \{\langle \mathsf{eval}_v(P_1)[i], \cdots, \mathsf{eval}_v(P_n)[i] \rangle : i \in [1, |\mathsf{ex}|\}$$

$$\mathsf{exT} = \{\mathsf{ex}[i] : \mathsf{eval}_v(P_{cond})[i] = \mathtt{T}\}$$

$$\mathsf{exF} = \{\mathsf{ex}[i] : \mathsf{eval}_v(P_{cond})[i] = \mathtt{F}\}$$

B Appendix

Integer Programs. For integer programs, we used the classic Fibonacci example `fib`, recursive modulo function `modulo`, recursive squaring of a number `square`, sum of all numbers less than n `sum_under`, and greatest common divisor `gcd`. Moreover, we were able to synthesize the classic mutually recursive program `iseven/isodd`, where `iseven` is synthesized with `isodd` is given as a component and vice-versa.

List Programs. For list programs, we used `stutter`, where each element in a list is duplicated, `compress`, where repeated consecutive elements of a given list are compressed into one, `insert`, where a given element e is inserted at a given position n of some list, list reversal `reverse`, recursive list concatenation `concat`, take/drop first/last n elements of a list `take/drop`, return last element of a list `last`.

Tree Programs. For binary tree manipulating programs, we used a function `count_leaves` that counts the leaves of the tree, a function `collect_leaves` that collects all leaves into a list, a function `nodes_at_level` that collects all nodes at level n of the tree into a list, and, finally, a function `hbal_tree` that constructs a balanced binary tree of size n.