Trace-Guided Inductive Synthesis of Recursive Functional Programs

YONGWEI YUAN, Purdue University, USA
ARJUN RADHAKRISHNA, Microsoft, USA
ROOPSHA SAMANTA, Purdue University, USA

We propose a novel trace-guided approach to tackle the challenges of ambiguity and generalization in synthesis of recursive functional programs from input-output examples. Our approach augments the search space of programs with recursion traces consisting of recursive subcalls of the programs. Our method is based on a new version space algebra (VSA) for succinct representation and efficient manipulation of pairs of recursion traces and programs that are consistent with each other. We have implemented this approach in a tool called SyRup and evaluated it on benchmarks from prior work. Our evaluation demonstrates that SyRup not only requires fewer examples to achieve a certain success rate than existing synthesizers, but is also less sensitive to the quality of the examples.

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

Additional Key Words and Phrases: Program Synthesis, Recursive Functional Programs

ACM Reference Format:
Yongwei Yuan, Arjun Radhakrishna, and Roopsha Samanta. 2023. Trace-Guided Inductive Synthesis of Recursive Functional Programs. *Proc. ACM Program. Lang.* 7, PLDI, Article 141 (June 2023), 24 pages. https://doi.org/10.1145/3591255

1 INTRODUCTION

The last decade has witnessed significant advances in synthesis of recursive functional programs [Albarghouthi et al. 2013; Farzan and Nicolet 2021; Feser et al. 2015; Itzhaky et al. 2021; Kneuss et al. 2013; Lee and Cho 2023; Lubin et al. 2020; Miltner et al. 2022; Osera and Zdancewic 2015; Polikarpova et al. 2016], and, in particular, in inductive synthesis of recursive programs from input-output (I/O) examples [Albarghouthi et al. 2013; Feser et al. 2015; Lubin et al. 2020; Miltner et al. 2022; Osera and Zdancewic 2015]. The latter problem, however, is far from solved and continues to pose several challenges that are best illustrated through an example. Let us consider a synthesis task where the goal is a recursive program that adds two natural numbers and is consistent with an I/O example $(2, 1) \rightsquigarrow 3$. In Fig. 1, we list several candidate programs in the search space of a typical synthesizer for functional programs, along with the corresponding *recursion traces*; informally, recursion traces capture sequences of recursive subcalls of a program induced by an I/O example. While all programs are consistent with the example $(2, 1) \rightsquigarrow 3$, notice that only the programs annotated with a ✓ are correct for all possible program inputs. So, how would the typical synthesizer resolve the ambiguity between these programs and pick one that generalizes to other inputs?

Authors' addresses: Yongwei Yuan, yuan311@purdue.edu, Purdue University, USA; Arjun Radhakrishna, Microsoft, USA, arradha@microsoft.com; Roopsha Samanta, Purdue University, USA, roopsha@purdue.edu.

2475-1421/2023/6-ART141
https://doi.org/10.1145/3591255

Recursion Trace	Program	

$(2,1)\hookrightarrow 3$
$\downarrow$
$(1,1)\hookrightarrow 2$
$\downarrow$
$(0,1)\hookrightarrow 1$
	`let rec nat_add m n =`
	`  match m with`
	`  | Zero -> n`
	`  | Succ m' ->`
	`  Succ (nat_add m' n)`
		✓

$(2,1)\hookrightarrow 3$
$\downarrow$
$(1,1)\hookrightarrow 3$
$\downarrow$
$(0,1)\hookrightarrow 3$
	`let rec nat_add1 m n =`
	`  match m with`
	`  | Zero -> Succ (Succ n)`
	`  | Succ m' ->`
	`  nat_add1 m' n`
		✗

$(2,1)\hookrightarrow 3$
$\downarrow$
$(1,0)\hookrightarrow 1$
	`let rec nat_add2 m n =`
	`  match m, n with`
	`  | Zero, Zero -> Zero`
	`  | Succ m', Zero -> m`
	`  | Zero, Succ n' -> n`
	`  | Succ m', Succ n' ->`
	`  Succ (Succ (nat_add2 m' n'))`
		✓

$(2,1)\hookrightarrow 3$	`let rec nat_add3 m n =` `  Succ (Succ n)`	✗

Fig. 1. Consistent trace-program pairs for an I/O example $(2,1)\hookrightarrow 3$.

— The synthesizer may use everyone's favorite inductive bias—*Occam's razor*—to pick the "smallest" program among the candidates [Barke et al. 2020; Gulwani 2011; Wang et al. 2017]. In our example, however, program `nat_add3` is the smallest and is obviously incorrect; further, programs `nat_add` and `nat_add1` are about the same size and only one of them is correct.
— Or, the synthesizer may require the user to provide a *complete* recursion trace, say $((2,1)\hookrightarrow 3 \longrightarrow (1,1)\hookrightarrow 2 \longrightarrow (0,1)\hookrightarrow 1)$, and use the recursion trace to find a program, such as `nat_add`, consistent with the trace [Albarghouthi et al. 2013; Feser et al. 2015; Osera and Zdancewic 2015]. This is an instance of programming by demonstration [Lau et al. 2003] and trace-based synthesis [Chugh et al. 2016], and helps the synthesizer by expecting more information about the behavior of the target program. Unfortunately, requiring users to provide a complete recursion trace, or, essentially, a *trace-complete* set of examples can be prohibitive.

In this paper, we propose a new *trace-guided* synthesis approach to tackle the challenges of trace-completeness, generalization, and ambiguity-resolution in inductive synthesis of recursive programs. Our approach augments the search space of programs with recursion traces, thereby using recursion traces as an inductive bias without burdening users. Given a set of I/O examples, our method *jointly* explores the space of recursion traces and programs that are consistent with each other, keeping track of consistent trace-program pairs in a version space [Lau et al. 2003; Mitchell 1982], and using a ranking function over such trace-program pairs to pick the most likely program.

The key technical contribution of this work is a new version space algebra (VSA) for succinct representation and efficient manipulation of consistent trace-program pairs. Our VSA has several important properties. First, it allows us to explore the space of recursion traces and programs *incrementally*. In each step, we leverage *angelic semantics* [Bodik et al. 2010; Floyd 1967] to track possible valuations of recursive subcalls that a consistent program might make in *the next* recursion step. Second, it allows us to construct a version space of consistent trace-program pairs *compositionally*. The VSA operators enable us to compose the version space for the current recursion step with version spaces for subsequent recursion steps through a clever manipulation that involves *consing* recursion traces and *unifying* sets of programs. Last, but not the least, all VSA operators are designed to be *consistency-preserving*, thereby eliminating the need for backtracking.

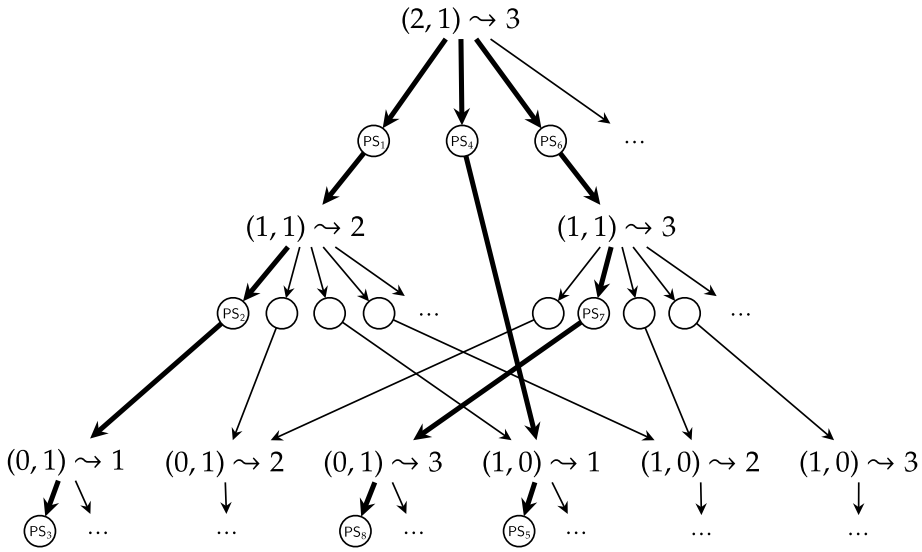

Fig. 2. An AND/OR graph representing a trace-indexed VS for I/O example $(2,1)\rightsquigarrow 3$. Each ◯ is an AND node representing a set of programs and each $i\rightsquigarrow o$ is an OR node representing a synthesis problem.

We have implemented our approach in a tool called SyRup[1] and evaluated it on 43 programming tasks collected from prior work [Osera and Zdancewic 2015]. Our experiments extensively evaluate SyRup's ability to generalize across different classes of I/O example sets. SyRup not only requires *fewer* examples to achieve a certain *success rate* than the two state-of-the-art inductive synthesizers Smyth [Lubin et al. 2020] and Burst [Miltner et al. 2022], but is also *less sensitive* to the *quality* of the examples.

To summarize, our paper makes the following contributions.

(1) We present a new trace-guided procedure for inductive synthesis of recursive programs to tackle the challenges of trace-completeness, generalization, and ambiguity-resolution . Our procedure is sound and complete modulo a notion of observational equivalence (Sec. 5).
(2) We define a new VSA for succinct representation and efficient manipulation of the space of recursion traces and programs that are consistent with each other (Sec. 4).
(3) We develop a tool SyRup and extensively evaluate it on synthesis tasks. We demonstrate that SyRup outperforms prior work in multiple aspects (Sec. 6, Sec. 7).

2 TRACE-GUIDED SYNTHESIS: ILLUSTRATIVE OVERVIEW

In this section, we illustrate key aspects of our trace-guided synthesis algorithm using our running example of synthesis from one I/O example $(2,1)\rightsquigarrow 3$.

Recursion Traces. Given an I/O example and a recursive program, a *recursion trace* (or, in short, *trace*) is a tree that witnesses the execution of the program on the given example. In particular, the nodes $i\rightsquigarrow o$ of a trace correspond to recursive subcalls with input i and output o, where the root node represents the given I/O example and the descendants of a node represent subsequent recursive subcalls made in the program execution. A trace is *linear* iff each node has at most one child. Given an I/O example $i\rightsquigarrow o$, we say that a *trace-program pair* $(\mathsf{Tr}, \mathsf{P})$ *is consistent* iff the nodes

[1] SyRup stands for "Synthesizer for Recursive Programs".

of the trace Tr witness the recursive calls necessary to complete the execution of P on input i as described above.

For instance, for the example $(2, 1) \rightsquigarrow 3$, each row in Fig. 1 presents a consistent trace-program pair, where each trace is linear.

Trace-Indexed Version Space. Our approach leverages (recursion) traces for inductive synthesis of recursive programs using a *trace-indexed version space (VS)*. Given an example $i \rightsquigarrow o$, a trace-indexed VS consists of pairs (Tr, PS) of traces and sets of programs such that for each program P in PS, the trace-program pair (Tr, P) is consistent.

To help illustrate how our method constructs a trace-indexed VS, we use a rooted AND/OR graph as shown in Fig. 2. Each (Tr, PS) pair in the trace-indexed VS for example $(2, 1) \rightsquigarrow 3$ corresponds to a rooted tree from the root OR node to leaf AND nodes in the AND/OR graph. For instance, `nat_add` and its trace in the first row of Fig. 1 corresponds to the leftmost path in the AND/OR graph (indicated using thick edges), and so on. Note that because each trace in this example is linear, the rooted tree corresponding to each (Tr, PS) pair in the trace-indexed VS is simply a path.

We now illustrate how our method constructs the trace-indexed VS for the leftmost path in Fig. 2. Each OR node $i \rightsquigarrow o$ represents a new synthesis problem, defined by an example $i \rightsquigarrow o$, with the root OR node representing the original synthesis problem defined by $(2, 1) \rightsquigarrow 3$. The children of an OR node $i \rightsquigarrow o$ are AND nodes (denoted by circles) and the children of an AND node are OR nodes. Each AND node represents a program set containing programs that that are consistent with the parent OR-node, *assuming* the children OR-nodes. For instance, each program P in the program set PS_1 is consistent with the original example $(2, 1) \rightsquigarrow 3$ given the assumption (that P is also consistent with) $(1, 1) \rightsquigarrow 2$. An example of a program in the program set PS_1 is as shown below:

```
let rec nat_add4 m n =
  match m with
  | Zero -> ??
  | Succ m' -> Succ (nat_add4 m' n)
```

where ?? can be substituted with any well-typed expression. This program belongs to program set PS_1 because given the assumption `nat_add 1 1 = 2`, we have `nat_add 2 1` equals `Succ ( nat_add 1 1)` equals 3, as expected.

Similarly, the program set PS_2 contains programs that are consistent with $(1, 1) \rightsquigarrow 2$, given the assumption $(0, 1) \rightsquigarrow 1$. In particular, the program set PS_2 also contains the partial program shown above; given the assumption `nat_add 0 1 = 1`, we have `nat_add 1 1` equals `Succ (nat_add 0 1)` equals 2, as expected.

Finally, the program set PS_3 contains programs that are consistent with $(0, 1) \rightsquigarrow 1$, and includes the following program that does not involve any recursive subcalls:

```
let rec nat_add5 m n =
  match m with
  | Zero -> n
  | Succ m' -> ??
```

where ?? can be substituted by any well-typed expression as before.

The (Tr, PS) pair for the leftmost path can now be constructed by "combining" the assumptions into the trace $(2, 1) \rightsquigarrow 3 \longrightarrow (1, 1) \rightsquigarrow 2 \longrightarrow (0, 1) \rightsquigarrow 1$, and by "intersecting" the program sets PS_1, PS_2, and PS_3. Notice that this intersection contains the correct program `nat_add` in Fig. 1.

By exhaustively exploring other rooted trees in the AND/OR graph, our method constructs a trace-indexed VS that consists of (Tr, PS) pairs corresponding to all rooted trees in the AND/OR graph by *splicing together assumptions and program sets*. In particular, our method uses operators defined in our version space algebra to do these manipulations efficiently.

Inferring Assumptions. The exploration of an AND/OR graph, and thus the construction of a trace-indexed VS, is driven by a procedure for inferring assumptions about the target program. This involves conjecturing new I/O examples that the target program is expected to be consistent with, based on the synthesis problem defined by the current OR node. Let's revisit the assumption `nat_add4 1 1 = 2` from earlier. To tackle the synthesis problem defined by $(2,1) \rightsquigarrow 3$, we not only search for candidate programs like `nat_add4`, but also infer the output of the recursive subcall `nat_add4 1 1` such that `Succ(nat_add4 1 1)` equals to 3. In this case, the inference is trivially done by "deconstructing" $3 \equiv$ `Succ(Succ(Succ(Zero)))`. In general, the inference may be more involved. Consider an alternative candidate program for the synthesis problem:

```
let rec nat_add5 m n =
  match m with
  | Zero -> ??
  | Succ m' -> plus_one (nat_add5 m' n)
```

Here, `plus_one` is a background function with the same semantics as the constructor `Succ`. The inverse semantics of background functions, particularly recursive ones, are not always readily available. Hence, we symbolically encode the semantics of `plus_one` in an SMT solver, and query the solver for the output $\alpha_{(1,1)}$ of the recursive subcall on $(1,1)$ such that `plus_one` $\alpha_{(1,1)}$ `= 3`.

Trace-Based Ranking Function. Given a trace-indexed VS for an I/O example, how do we pick programs that generalize better? As argued in Sec. 1, simply choosing a program with the "smallest" size may not always suffice. Fortunately, the recursion traces in the trace-indexed VS encode rich *semantic* features of the corresponding programs. For instance, one may easily identify the recursion scheme used in a program from the shape of its recursion trace; if we simply prefer programs with longer traces, then the program `nat_add` ranks higher than the smaller non-recursive program `nat_add3` in Fig. 1.

3 PROBLEM DEFINITION AND SOLUTION OVERVIEW

$$
\begin{array}{llll}
\text{program} & \mathsf{P} & ::= & \texttt{fix}\ \mathsf{f}(x_{\mathsf{in}}).e \\
\text{expression} & e, g & ::= & x \mid e_1\ (e_2) \mid C() \mid C(e) \\
 & & \mid & \texttt{match}\ x\ \texttt{with}\ \{C_1(x_1) \to e_1, \ldots, C_{\mathsf{n}}(x_{\mathsf{n}}) \to e_{\mathsf{n}}\} \\
 & & \mid & \texttt{cond}\ \{g_1 \to e_1, \ldots, g_{\mathsf{n}} \to e_{\mathsf{n}}\} \\
\text{value} & \mathsf{i}, \mathsf{o}, v & ::= & \texttt{true} \mid \texttt{false} \mid C() \mid C(v)
\end{array}
$$

$$
\frac{e_2 \Downarrow v_2 \qquad [x_{\mathsf{in}} := v_2, \mathsf{f} := (\texttt{fix}\ \mathsf{f}(x_{\mathsf{in}}).e_1)]\ e_1 \Downarrow v}{(\texttt{fix}\ \mathsf{f}(x_{\mathsf{in}}).e_1)\ (e_2) \Downarrow v}\ \textsc{ERec} \qquad \frac{e \Downarrow v}{C(e) \Downarrow C(v)}\ \textsc{ECtor}
$$

$$
\frac{g \Downarrow \texttt{true} \qquad e \Downarrow v}{\texttt{cond}\ \{\ldots g \to e \ldots\} \Downarrow v}\ \textsc{ECond} \qquad \frac{e' \Downarrow C(v') \qquad [x := v']\ e \Downarrow v}{\texttt{match}\ e'\ \texttt{with}\ \{\ldots C(x) \to e \ldots\} \Downarrow v}\ \textsc{EMatch}
$$

Fig. 3. tinyML: Syntax and Semantics.

Algorithm 1 Top-Level Synthesis Algorithm

Input: I/O examples $\mathcal{E}$
Input: Layered search space $(\mathcal{L}_j)_{j\in\mathbb{N}}$
Output: Program P

```
1: procedure SYNTHESIZE(E, (L_j)_{j∈ℕ})
2:     for j ∈ ℕ do
3:         Ψ ← LEARN(E, L_j)
4:         if Ψ ≠ ∅ then
5:             P ← RANK(Ψ)
6:             return P
```

Given a set $\mathcal{E}$ of I/O examples, our goal is to synthesize a recursive functional program P of the form `fix` $f(x_{in}).e$ in a search space defined by a DSL. In what follows, we present our target DSL, formalize our problem statement, and present our top-level synthesis algorithm.

Our target DSL tinyML is a functional ML-like programming language with recursive functions, algebraic data types, and pattern matching (see Fig. 3). The syntax of expressions is mostly standard, with variables x, function applications e_1 (e_2), and constructors C. Both function applications and constructors may take multiple arguments (omitted in Fig. 3). To ensure that programs terminate, recursive subcalls are only permitted on values that are strictly smaller than the input. Expressions include a restricted form of `match`, where scrutinees can only be variables, and patterns are required to be non-overlapping and exhaustive, i.e., there is exactly one branch for each constructor of the data type being matched. Finally, expressions include conditional expressions `cond` consisting of a set of *guarded* expressions, where guards are essentially Boolean expressions that do not permit recursive subcalls to f. The guards in a `cond` expression are required to be mutually-exclusive and exhaustive; this ensures that `cond` expressions are equivalent in expressiveness to the more commonly used `if` expressions. The program semantics is standard.

Definition 3.1 (Program Correctness)**.** Given a program P in tinyML, P is consistent w.r.t. a set $\mathcal{E}$ of I/O examples, denoted $P \models \mathcal{E}$, if and only if $P\ (i) \Downarrow o$ for each I/O example $i \rightsquigarrow o \in \mathcal{E}$.

Let us denote by $\mathcal{L}$ the set of programs generated by tinyML.

Definition 3.2 (Problem Definition)**.** Given a set $\mathcal{E}$ of I/O examples, find a program P such that $P \in \mathcal{L}$ and $P \models \mathcal{E}$.

Our top-level synthesis procedure is presented in Algo. 1. Because $\mathcal{L}$ is unbounded, we follow common practice in program synthesis literature to both bound the search space and prioritize search. Thus, we assume that $\mathcal{L}$ is *layered* into a sequence of finite subspaces $(\mathcal{L}_1, \mathcal{L}_2, \ldots)$, where each subspace $\mathcal{L}_j$ is obtained from $\mathcal{L}$ by constraining the value of some suitable program parameter(s), and the ordering of subspaces encodes preference based on parameter values. For instance, a popular layering strategy is to use *program size* to bound the search space and prefer *smaller* programs by ordering subspaces by increasing size.

Definition 3.3 (Layered Search Space[2])**.** Search space $\mathcal{L}$ is defined to be *layered* by a sequence $(\mathcal{L}_1, \mathcal{L}_2, \ldots)$ of finite subsets $\mathcal{L}_j$ of $\mathcal{L}$, denoted $\mathcal{L} = (\mathcal{L}_j)_{j\in\mathbb{N}}$, iff $\mathcal{L} = \bigcup_{j\in\mathbb{N}} \mathcal{L}_j$.

Given a set $\mathcal{E}$ of examples and a layered search space $\mathcal{L} = (\mathcal{L}_j)_{j\in\mathbb{N}}$, procedure SYNTHESIZE (1) iterates through each finite subspace $\mathcal{L}_j$ of $\mathcal{L}$, (2) learns a (representation of a) set Ψ of programs

[2]Notice that, unlike conventional notions of layering, say based on program size, we do not require $\mathcal{L}_{j+1} \supseteq \mathcal{L}_j$. As we shall see in Sec. 5 and Sec. 7, our relaxed notion of layering offers more flexibility in exploring the search space $\mathcal{L}$.

in subspace $\mathcal{L}_j$ such that $P \models \mathcal{E}$ for each $P \in \Psi$, and (3) returns a program in Ψ that maximizes a ranking function on $\mathcal{L}$. In any iteration, if Ψ is empty, the procedure moves onto the next subspace.

Most existing solutions [Barke et al. 2020; Lee 2021; Lubin et al. 2020; Miltner et al. 2022] for the program synthesis problem in Def. 3.3 are instances of Algo. 1 and iteratively increase the upper bound on some program parameter (such as AST size or depth) to systematically expand the search space. The key distinguishing elements lie in the design of the layered search space, the Learn procedure, and the ranking function.

In what follows, we formalize our version space algebra to represent and manipulate sets of programs in Sec. 4. We then present the definition of our layered search space along with the Learn and Rank procedures for our program representation in Sec. 5 and Sec. 6 respectively.

4 TRACE-INDEXED EXTENDED VERSION SPACES

In this section, we present the definitions required to describe our synthesis algorithm.

4.1 Trace-Witnessed Semantics

Frames and Recursion Traces. We use *frames* to represent the behavior within the current instance of a call to the recursive function f and *recursion traces* to represent the behavior of the full tree of recursive sub-calls. A frame Fr is denoted as $(\Delta \rightsquigarrow v) \rightarrow \text{Asm}$ where: (a) Δ is a sequence of substitutions $x_j := v_j$, called valuation context, with $[\Delta]\, e$ being the result of applying substitutions in e, (b) v is a target value, and (c) $\text{Asm} = [\mathsf{i}_1 \rightsquigarrow \mathsf{o}_1 \;\; \ldots \;\; \mathsf{i}_n \rightsquigarrow \mathsf{o}_n]$ is an *assumption sequence* of I/O examples $\mathsf{i}_j \rightsquigarrow \mathsf{o}_j$ where we also have that $\mathsf{i}_j = \mathsf{i}_k \implies \mathsf{o}_j = \mathsf{o}_k$. A recursion trace Tr is given by a tree $(\mathsf{i} \rightsquigarrow \mathsf{o}) \rightarrow \text{Trs}$ where: (a) the root Root(Tr) is an I/O example $(\mathsf{i} \rightsquigarrow \mathsf{o})$, and (b) the (possibly empty) sequence of recursion traces $\text{Trs} = [\text{Tr}_1 \;\; \ldots \;\; \text{Tr}_n]$ are the children. We assume that the set of I/O examples in a recursion trace is consistent, i.e., no two nodes are labeled with examples having the same input but differing outputs.

For brevity, frame $((x_{\text{in}} := \mathsf{i}) \rightsquigarrow \mathsf{o}) \rightarrow \text{Asm}$ is written as $(\mathsf{i} \rightsquigarrow \mathsf{o}) \rightarrow \text{Asm}$ for the rest of the paper.

Example 4.1. Consider the tinyML program $P \equiv (\texttt{fix}\ \mathsf{f}(x_{\text{in}}).e)$ where $e \equiv \texttt{match}\ x_{\text{in}}\ \texttt{with}\ \{\texttt{nil} \rightarrow 0, \texttt{cons}(\mathsf{h}, \mathsf{t}) \rightarrow \mathsf{h} + \mathsf{f}(\mathsf{t})\}$. On the input $\mathsf{i} = \texttt{cons}(5, \texttt{cons}(10, \texttt{nil}))$, the program P returns 15. The invocation to f(i) makes one direct recursive sub-call (i.e., $\mathsf{f}(\texttt{cons}(10, \texttt{nil}))$), which in turn makes another recursive sub-call $\mathsf{f}(\texttt{nil})$. This behavior is represented by the frame $\text{Fr} = (\mathsf{i} \rightsquigarrow 15) \rightarrow [\texttt{cons}(10, \texttt{nil}) \rightsquigarrow 10]$ and the recursion trace $\text{Tr} = (\mathsf{i} \rightsquigarrow 15) \rightarrow [(\texttt{cons}(10, \texttt{nil}) \rightsquigarrow 10) \rightarrow [(\texttt{nil} \rightsquigarrow 0) \rightarrow []]]$. Here, the frame Fr captures the local behavior, i.e., that calling f on i returns 15 while making one direct recursive sub-call characterized by $\texttt{cons}(10, \texttt{nil}) \rightsquigarrow 10$; while the recursion trace Tr characterizes the full nested and thus global recursion behavior. □

Example 4.2. Consider a standard recursive implementation e^* for computing Fibonacci Numbers: $e^* \equiv \texttt{cond}\ \{x_{\text{in}} = 0 \rightarrow 1, x_{\text{in}} = 1 \rightarrow 1, x_{\text{in}} > 1 \rightarrow \mathsf{f}(x_{\text{in}} - 2) + \mathsf{f}(x_{\text{in}} - 1)\}$. Similarly, the recursive sub-call $(\texttt{fix}\ \mathsf{f}(x_{\text{in}}).e^*)\ (3)$ is characterized by the frame $(3 \rightsquigarrow 3) \rightarrow [1 \rightsquigarrow 1, 2 \rightsquigarrow 2]$ and the trace $\text{Tr}^* = (3 \rightsquigarrow 3) \rightarrow [(1 \rightsquigarrow 1) \rightarrow [], (2 \rightsquigarrow 2) \rightarrow [(0 \rightsquigarrow 1) \rightarrow [], (1 \rightsquigarrow 1) \rightarrow []]]$. Note that Tr^* (unlike Tr from Example 4.1) is branching, i.e., that a node has more than 1 child corresponding to multiple direct recursive sub-calls. □

Angelic Semantics and Trace-Witnessed Semantics. The angelic semantics and trace-witnessed semantics formalize the idea of frames and traces representing local and global recursion behavior as seen in Example 4.1. The rules for angelic and trace-witnessed semantics are shown in Fig. 4. Intuitively, the angelic semantics evaluate a term left-to-right in a standard manner, except that when a recursive function call $\mathsf{f}(e)$ is encountered, its output value is obtained from the assumption sequence Asm. Formally, the judgment $e \Downarrow^{\text{Asm}}_{\text{Asm}'} v$ asserts that: (a) evaluating the expression e requires $n = |\text{Asm}| - |\text{Asm}'|$ direct recursive calls, (b) the arguments to the recursive calls and their

$$\frac{e \Downarrow v}{e \Downarrow^{\mathsf{Asm}}_{\mathsf{Asm}} v}\ \textsc{AEBase} \qquad \frac{g \Downarrow^{\mathsf{Asm}}_{\mathsf{Asm}'} \mathtt{true} \qquad e \Downarrow^{\mathsf{Asm}'}_{\mathsf{Asm}''} v}{\mathtt{cond}\ \{\ldots g \to e \ldots\} \Downarrow^{\mathsf{Asm}}_{\mathsf{Asm}''} v}\ \textsc{AECond}$$

$$\frac{e \Downarrow^{\mathsf{Asm}}_{\mathsf{Asm}'} v}{C(e) \Downarrow^{\mathsf{Asm}}_{\mathsf{Asm}'} C(v)}\ \textsc{AECtor} \qquad \frac{e \Downarrow^{\mathsf{Asm}}_{\mathsf{Asm}'} C(v) \qquad [x := v]\, e \Downarrow^{\mathsf{Asm}'}_{\mathsf{Asm}''} v}{\mathtt{match}\ e\ \mathtt{with}\ \{\ldots C(x) \to e \ldots\} \Downarrow^{\mathsf{Asm}}_{\mathsf{Asm}''} v}\ \textsc{AEMatch}$$

$$\frac{e \Downarrow^{\mathsf{Asm}}_{\mathsf{Asm}'} \mathsf{i} \qquad \mathsf{Asm}' = [\mathsf{i} \rightsquigarrow \mathsf{o}] \sqcup \mathsf{Asm}''}{\mathsf{f}\,(e) \Downarrow^{\mathsf{Asm}}_{\mathsf{Asm}''} \mathsf{o}}\ \textsc{AERec} \qquad \frac{\begin{array}{c}\mathsf{Asm} = [\ldots \quad \mathsf{Root}(\mathsf{Tr}_j) \quad \ldots] \\ [x := \mathsf{i}]\, e \Downarrow^{\mathsf{Asm}}_{[]} \mathsf{o} \qquad \forall \mathsf{j}.\ (\mathtt{fix}\ \mathsf{f}(x).e) \models \mathsf{Tr}_j\end{array}}{(\mathtt{fix}\ \mathsf{f}(x).e) \models ((\mathsf{i} \rightsquigarrow \mathsf{o}) \to [\ldots \quad \mathsf{Tr}_j \quad \ldots])}\ \textsc{TERec}$$

Fig. 4. Angelic semantics and trace-witnessed semantics of a recursive program $\mathsf{P} = \mathtt{fix}\ \mathsf{f}(x).e$.

return values form the first n I/O examples in Asm, and (c) Asm′ is the suffix of Asm excluding the first n I/O examples. And we write Asm \ Asm′ to denote the assumption sequence obtained from removing the suffix Asm′ from Asm.

The trace-witnessed semantics build on the angelic semantics. Intuitively, Rule TERec combines the local behavior (the 2^{nd} premise stating that f angelically evaluates i to o making assumptions Asm) and the global behavior (the 3^{rd} premise stating that for each assumption $\mathsf{i}_j \rightsquigarrow \mathsf{o}_j$ in Asm, there is a trace Tr_j witnessing it).

Example 4.3. Consider the program P, frame $\mathsf{Fr} = (\mathsf{i} \rightsquigarrow 15) \to \mathsf{Asm}$, and Tr from Example 4.1. Using the angelic semantics, we can derive $e \Downarrow^{\mathsf{Asm}}_{[]} 15$, i.e., assuming $\mathsf{f}\,(\mathtt{cons}(10, \mathtt{nil}))$ evaluates to 10, we have that e evaluates to 15. Note that the value of the recursive call is not obtained by executing e on the arguments, but from the assumption list $[\mathtt{cons}(10, \mathtt{nil}) \rightsquigarrow 10]$. In fact, we also have $e' \Downarrow^{[\mathtt{cons}(10,\mathtt{nil}) \rightsquigarrow 10]}_{[]} 15$ for $e' \equiv \mathtt{match}\ x_{\mathsf{in}}\ \mathtt{with}\ \{\mathtt{nil} \to 100, \mathsf{cons}(\mathsf{h}, \mathsf{t}) \to \mathsf{h} + \mathsf{f}\,(\mathsf{t})\}$. But we cannot derive $(\mathtt{fix}\ \mathsf{f}(x_{\mathsf{in}}).e') \models \mathsf{Tr}'$ for any $\mathsf{Tr}' = (\mathsf{cons}(5, \mathsf{cons}(10, \mathtt{nil})) \rightsquigarrow 15) \to \mathsf{Trs}$, while we can derive $(\mathtt{fix}\ \mathsf{f}(x_{\mathsf{in}}).e) \models \mathsf{Tr}$ for Tr defined in Example 4.1. □

Thm. 4.4 establishes the equivalence of the trace-witnessed semantics and standard semantics of programs in tinyML with the help of Lem. 4.5 and Lem. 4.6.

Theorem 4.4 (Soundness and Completeness of Trace-Witnessed Semantics)**.** Given an I/O example $\mathsf{i} \rightsquigarrow \mathsf{o}$, we have that $\mathsf{P}\,(\mathsf{i}) \Downarrow \mathsf{o}$ if and only if $\mathsf{P} \models \mathsf{Tr}$ for some trace $\mathsf{Tr} = (\mathsf{i} \rightsquigarrow \mathsf{o}) \to \mathsf{Trs}$.

Proof. Soundness follows by rule induction on $\mathsf{P} \models \mathsf{Tr}$. The only case that applies is TERec. We have $\mathsf{P} = (\mathtt{fix}\ \mathsf{f}(x_{\mathsf{in}}).e)$, $\mathsf{Tr} = (\mathsf{i} \rightsquigarrow \mathsf{o}) \to [\ldots \quad \mathsf{Tr}_j \quad \ldots]$, $[x_{\mathsf{in}} := \mathsf{i}]\, e \Downarrow^{\mathsf{Asm}}_{[]} \mathsf{o}$, and $(\mathtt{fix}\ \mathsf{f}(x_{\mathsf{in}}).e) \models \mathsf{Tr}_j$ for all j. Let's say $\mathsf{Root}(\mathsf{Tr}_j) = \mathsf{i}_j \rightsquigarrow \mathsf{o}_j$ for all j, and $\mathsf{Asm} = [\ldots \quad \mathsf{i}_j \rightsquigarrow \mathsf{o}_j \quad \ldots]$. By the inductive hypothesis, $\mathsf{P}\,(\mathsf{i}_j) \Downarrow \mathsf{o}_j$. Then by Lem. 4.5 (see below), $\mathsf{P}\,(\mathsf{i}) \Downarrow \mathsf{o}$. Lastly, completeness follows directly from Lem. 4.6 (see below). □

Lemma 4.5. If $e \Downarrow^{\mathsf{Asm}}_{\mathsf{Asm}'} v$ and $\mathsf{P}\,(\mathsf{i}) \Downarrow \mathsf{o}$ for all $\mathsf{i} \rightsquigarrow \mathsf{o} \in \mathsf{Asm} \setminus \mathsf{Asm}'$ then $e \Downarrow v$.

Proof. By rule induction on $e \Downarrow^{\mathsf{Asm}}_{\mathsf{Asm}'} v$. Since the assumptions used in evaluation of sub-expression(s) have to be subsequences of Asm \ Asm′, applying the inductive hypothesis gives us the evaluation of sub-expression(s) in standard semantics. Applying rules from standard semantics, we reconstruct the evaluation of e in standard semantics. □

Lemma 4.6. If $e \Downarrow v$ then $\begin{cases} e_1 \Downarrow v_1 \text{ and } \mathsf{P} \models (v_1 \rightsquigarrow v) \rightarrow \mathsf{Trs} \text{ for some } \mathsf{Trs} & \text{if } e = \mathsf{P}\,(e_1) \\ e \Downarrow^{\mathsf{Asm}_1}_{\mathsf{Asm}_2} v \text{ for some } \mathsf{Asm}_2 \text{ for arbitrary } \mathsf{Asm}_1 \text{ and} \\ \mathsf{P} \models (\mathsf{i}_\mathsf{j} \rightsquigarrow \mathsf{o}_\mathsf{j}) \rightarrow \mathsf{Trs}_\mathsf{j} \text{ for some } \mathsf{Trs}_\mathsf{j} & \text{otherwise} \\ \quad \text{for all } \mathsf{i}_\mathsf{j} \rightsquigarrow \mathsf{o}_\mathsf{j} \in \mathsf{Asm}_1 \setminus \mathsf{Asm}_2 \end{cases}$

Proof. By rule induction on $e \Downarrow v$ and case analysis on whether the sub-expression(s) in e is a recursive subcall or not, we apply the inductive hypothesis accordingly and reconstruct the evaluation of e in trace-witnessed semantics along with witnessing Asm and/or Tr. □

4.2 Extended Version Spaces

Version space algebras (VSAs) are a way to compactly represent and manipulate solution sets to synthesis problems [Gulwani 2011; Lau et al. 2003; Polozov and Gulwani 2015]. In this section, we extend this representation with a new kind of version space operator called *choice* which allows us to more efficiently represent solution sets that involve conditional operators.

An *extended version space* is given by a term specified by the syntax below.

$$\mathsf{PS} ::= \top \mid \{e_1, \ldots, e_\mathsf{n}\} \mid \cup\,(\mathsf{PS}_1, \ldots, \mathsf{PS}_\mathsf{n}) \mid \bowtie_{\mathsf{op}}\,(\mathsf{PS}_1, \ldots, \mathsf{PS}_\mathsf{n}) \mid \boldsymbol{\Pi}\,(\mathsf{PS}_1, \ldots, \mathsf{PS}_\mathsf{n})$$

Here, PS_j are extended version spaces, e_j are tinyML expressions, and op is a function, a constructor, or a `match` from tinyML. The first four operators (top, list, union, and join) are standard version space algebra operators and have standard semantics. More formally, given any tinyML expression e, we have that: (a) $e \in \top$, (b) $e \in \{e_1, \ldots, e_\mathsf{n}\}$ if $\exists \mathsf{j}.e = e_\mathsf{j}$, (c) $e \in \cup\,(\mathsf{PS}_1, \ldots, \mathsf{PS}_\mathsf{n})$ if $\exists \mathsf{j}.e \in \mathsf{PS}_\mathsf{j}$, and (d) $e \in \bowtie_{\mathsf{op}}\,(\mathsf{PS}_1, \ldots, \mathsf{PS}_\mathsf{n})$ if e is of the form $\mathsf{op}(e_1, \ldots, e_\mathsf{n})$ and $\forall \mathsf{j}.e_\mathsf{j} \in \mathsf{PS}_\mathsf{j}$. Note that these semantics do not cover conditional terms, i.e., those involving the operator `cond` {}.

To understand how conditional terms are handled and how they interact with the choice operator $\boldsymbol{\Pi}\,(\cdot)$, we first define the notion of *supporting valuations*. Formally, each extended version space PS is implicitly annotated with a set of valuations denoted by $\mathsf{Support}(\mathsf{PS})$. Intuitively, in our synthesis procedures from Sec. 5, if PS represents the solution set to a synthesis task, any program P that behaves equivalent to some program $\mathsf{P}^* \in \mathsf{PS}$ on all valuations in $\mathsf{Support}(\mathsf{PS})$ is also a solution. Our procedures ensure that a version space rooted at a non-choice operator has the same support as its sub-spaces, while the support of a version space rooted at a choice operator is the *disjoint* union of the support of its sub-spaces.

Example 4.7. Consider the version spaces $\mathsf{PS}_1 \equiv \{x_\mathsf{in}, 2 \times x_\mathsf{in}\}$ with support $\mathsf{Support}(\mathsf{PS}_1) = \{x_\mathsf{in} := 1, x_\mathsf{in} := 2\}$ and $\mathsf{PS}_2 \equiv \{-x_\mathsf{in}\}$ with support $\mathsf{Support}(\mathsf{PS}_2) = \{x_\mathsf{in} := -1, x_\mathsf{in} := 0\}$. These two version spaces are disjoint in the classical sense. However, the program $\mathsf{P}^* \equiv \mathsf{cond}\ \{x_\mathsf{in} > 0 \rightarrow x_\mathsf{in}, x_\mathsf{in} \leq 0 \rightarrow -x_\mathsf{in}\}$ behaves like a program from both PS_1 and PS_2 on their respective supports. □

Thus, the choice operator $\boldsymbol{\Pi}\,(\cdot)$ in *extended version spaces* enables the inclusion of programs like P^* in $\boldsymbol{\Pi}\,(\mathsf{PS}_1, \mathsf{PS}_2)$. Formally, given a tinyML expression $e \equiv \mathsf{cond}\ \{\ldots, g_\mathsf{k} \rightarrow e_\mathsf{k}, \ldots\}$ and an extended version space $\mathsf{PS} \equiv \boldsymbol{\Pi}\,(\ldots, \mathsf{PS}_\mathsf{j}, \ldots)$, we have $e \in \mathsf{PS}$ if

$$\forall \mathsf{j}.\exists \mathsf{k}.\ \left(e_\mathsf{k} \in \mathsf{PS}_\mathsf{j} \wedge \forall \Delta \in \mathsf{Support}(\mathsf{PS}_\mathsf{j}).[\Delta] g_\mathsf{k} \Downarrow \mathsf{true}\right)$$

Combining the rule with the standard rules from the previous paragraph gives the full semantics of extended version spaces.

Example 4.8. Continuing from Example 4.7, the program P^* is contained in $\boldsymbol{\Pi}\,(\mathsf{PS}_1, \mathsf{PS}_2)$, i.e., $\mathsf{P}^* \in \boldsymbol{\Pi}\,(\mathsf{PS}_1, \mathsf{PS}_2)$. This is because for both PS_1 and PS_2, there exists a unique matching branch in P^* whose guard is satisfied by any valuations in $\mathsf{Support}(\mathsf{PS}_1)$ and $\mathsf{Support}(\mathsf{PS}_2)$ respectively, i.e., $(x_\mathsf{in} > 0)$ is satisfied by $\{x_\mathsf{in} := 1, x_\mathsf{in} := 2\}$, and $(x_\mathsf{in} \leq 0)$ is satisfied by $\{x_\mathsf{in} := -1, x_\mathsf{in} := 0\}$. □

We now define the extended version of the version space intersection operation. The intersection of two extended version spaces is defined by the following rules (modulo symmetry):

$$
\begin{array}{l}
\textcircled{1}\ \top \cap \mathsf{PS} = \mathsf{PS} \\
\textcircled{2}\ \{\mathsf{PS}\} \cap \{\mathsf{PS}\} = \{\mathsf{PS}\} \\
\textcircled{3}\ \{\mathsf{PS}_1, \ldots, \mathsf{PS}_n\} \cap \mathsf{PS} = \{\mathsf{PS}_1 \cap \mathsf{PS}, \ldots, \mathsf{PS}_n \cap \mathsf{PS}\} \\
\textcircled{4}\ \cup(\mathsf{PS}_1, \ldots, \mathsf{PS}_n) \cap \mathsf{PS} = \cup(\mathsf{PS}_1 \cap \mathsf{PS}, \ldots, \mathsf{PS}_n \cap \mathsf{PS}) \\
\textcircled{5}\ \bowtie_{\mathsf{op}} \left(\mathsf{PS}'_1, \ldots, \mathsf{PS}'_k\right) \cap \bowtie_{\mathsf{op}} \left(\mathsf{PS}''_1, \ldots, \mathsf{PS}''_k\right) = \bowtie_{\mathsf{op}} \left(\mathsf{PS}'_1 \cap \mathsf{PS}''_1, \ldots, \mathsf{PS}'_k \cap \mathsf{PS}''_k\right) \\
\textcircled{6}\ \mathsf{PS}_1 \cap \mathsf{PS}_2 = \mathbf{\Pi}(\mathsf{PS}_1, \mathsf{PS}_2, \cup(\mathsf{PS}_1, \mathsf{PS}_2)) \qquad \text{if rules } \textcircled{1}\text{-}\textcircled{5} \text{ do not apply}
\end{array}
$$

Rules ①-② are the base cases. Rules ③-⑤ push intersections through the list, union, and join operators whenever possible. This is akin to how intersection is defined on standard version spaces. However, when these rules fail, we have a special rule ⑦ that produces a choice node $\mathbf{\Pi}(\mathsf{PS}_1, \mathsf{PS}_2, \cup(\mathsf{PS}_1, \mathsf{PS}_2))$. Here, if S_1 and S_2 are the supports of the inputs PS_1 and PS_2, the 3 operands of the choice operator have disjoint supports $S_1 \setminus S_2$, $S_2 \setminus S_1$, and $S_2 \cap S_1$, respectively. Thus, the intersection contains conditional programs (e.g. P^*) that "behave" like PS_1 on $S_1 \setminus S_2$, like PS_2 on $S_2 \setminus S_1$, and like one of PS_1 and PS_2 on $S_1 \cap S_2$. We use this notion of *observational equivalence* to formalize the soundness and completeness of extended version space intersection in the following theorem. The notation $\mathsf{P}_1 \equiv_S \mathsf{P}_2$ denotes that P_1 always evaluates to the same output as P_2 for the valuations in support S.

Theorem 4.9 (Observational Soundness and Completeness of Extended Version Space Intersection)**.** Given extended version spaces PS_1 and PS_2 with corresponding supports $\mathsf{Support}(\mathsf{PS}_1)$ and $\mathsf{Support}(\mathsf{PS}_2)$, their intersection $\mathsf{PS}_1 \cap \mathsf{PS}_2$ (as defined by the above rules) satisfies: (a) $\mathsf{P} \in \mathsf{PS}_1 \cap \mathsf{PS}_2 \implies (\mathsf{P} \in \mathsf{PS}_1 \wedge \mathsf{P} \in \mathsf{PS}_2) \vee (\forall \mathsf{j} \in \{1,2\}. \forall \mathsf{P}_\mathsf{j} \in \mathsf{PS}_\mathsf{j}. \mathsf{P}_\mathsf{j} \equiv_{\mathsf{Support}(\mathsf{PS}_\mathsf{j})} \mathsf{P})$ (b) $\mathsf{P} \in \mathsf{PS}_1 \wedge \mathsf{P} \in \mathsf{PS}_2 \implies \mathsf{P} \in \mathsf{PS}_1 \cap \mathsf{PS}_2$, and (c) $\mathsf{Support}(\mathsf{PS}_1 \cap \mathsf{PS}_2) = \mathsf{Support}(\mathsf{PS}_1) \cup \mathsf{Support}(\mathsf{PS}_2)$.

4.3 Indexed Extended Version Spaces

An *indexed extended version space (indexed VS)* $\Psi = \{\iota_1 \mapsto \mathsf{PS}_1, \ldots, \iota_\mathsf{n} \mapsto \mathsf{PS}_\mathsf{n}\}$ is a mapping from distinct indices ι_j to extended version spaces PS_j. In general, the indices represent some semantic information related to the expressions $e \in \mathsf{PS}$. More specifically, fixing the type of indices to be one of frames, recursion traces, or recursion trace sequences yields frame-indexed, trace-indexed or traces-indexed VSes, respectively. These are usually denoted using the symbols $\Psi^{\mathbf{fr}}$, $\Psi^{\mathbf{tr}}$, and $\Psi^{\mathbf{trs}}$, respectively.

We define notions of consistency for indexed version spaces that state that the programs in the version space "match" the semantics of the index. Formally,

(a) a frame-indexed VS $\Psi^{\mathbf{fr}}$ is consistent if $\forall((\Delta \rightsquigarrow v) \rightarrow \mathsf{Asm}) \mapsto \mathsf{PS} \in \Psi^{\mathbf{fr}}. \forall e \in \mathsf{PS}. [\Delta]e \Downarrow^{\mathsf{Asm}}_{[]} v$;

(b) a trace-indexed VS $\Psi^{\mathbf{tr}}$ is consistent if $\forall \mathsf{Tr} \mapsto \mathsf{PS} \in \Psi^{\mathbf{tr}}. \forall e \in \mathsf{PS}. (\texttt{fix}\ \mathsf{f}(x).e) \models \mathsf{Tr}$; and

(c) a traces-indexed VS $\Psi^{\mathbf{trs}}$ is consistent if $\forall \mathsf{Trs} \mapsto \mathsf{PS} \in \Psi^{\mathbf{trs}}. \forall e \in \mathsf{PS}, \mathsf{Tr} \in \mathsf{Trs}. (\texttt{fix}\ \mathsf{f}(x).e) \models \mathsf{Tr}$.

Now, we define operations on trace-indexed VSes, traces-indexed VSes, and frame-indexed VSes in Fig. 5, which preserve the consistency of indexed extended version spaces before and after the operations. Note that one may easily generalize definitions in Fig. 5 to handle more than 2 arguments.

- *Concat operation.* The Concat operation takes n consistent trace-indexed VSes $\Psi^{\mathbf{tr}}_\mathsf{j}$ whose indices are all distinct, and returns a trace-indexed VS containing all $\mathsf{Tr} \mapsto \mathsf{PS}$ mappings from each $\Psi^{\mathbf{tr}}_\mathsf{j}$. The resulting trace-indexed VS is trivially consistent.
- *Prod operation.* The Prod operation takes a consistent traces-indexed VS $\Psi^{\mathbf{trs}}$, and a consistent trace-indexed VS $\Psi^{\mathbf{tr}}$. For each pair of their mappings , we computes VS intersection while

$$\textsc{Concat}(\Psi^{\mathbf{tr}}_1, \ldots, \Psi^{\mathbf{tr}}_n) \stackrel{\text{def}}{=} \{\mathsf{Tr} \mapsto \mathsf{PS} \mid \exists 1 \le j \le n\,,\ (\mathsf{Tr} \mapsto \mathsf{PS}) \in \Psi^{\mathbf{tr}}_j\} \quad \text{where all Tr are distinct}$$

$$\textsc{Prod}(\Psi^{\mathbf{trs}}, \Psi^{\mathbf{tr}}) \stackrel{\text{def}}{=} \{\ \mathsf{Trs} \sqcup [\mathsf{Tr}] \mapsto \mathsf{PS}_1 \cap \mathsf{PS}_2 \mid (\mathsf{Trs} \mapsto \mathsf{PS}_1) \in \Psi^{\mathbf{trs}}, (\mathsf{Tr} \mapsto \mathsf{PS}_2) \in \Psi^{\mathbf{tr}}\}$$

$$\textsc{Cons}(\mathsf{Fr} \mapsto \mathsf{PS}, \Psi^{\mathbf{trs}}) \stackrel{\text{def}}{=} \{\ ((\mathsf{i} \rightsquigarrow \mathsf{o}) \to \mathsf{Trs}) \mapsto \mathsf{PS} \cap \mathsf{PS}' \mid (\mathsf{Trs} \mapsto \mathsf{PS}') \in \Psi^{\mathbf{trs}}\}$$
$$\text{where } \mathsf{Fr} = (\mathsf{i} \rightsquigarrow \mathsf{o}) \to \mathsf{Asm} \text{ and } \forall (\mathsf{Trs} \mapsto \mathsf{PS}) \in \Psi^{\mathbf{trs}}.\mathtt{Roots}(\mathsf{Trs}) = \mathsf{Asm}$$

$$\begin{aligned}\textsc{Merge}(\Psi^{\mathbf{fr}}_1, \Psi^{\mathbf{fr}}_2) \stackrel{\text{def}}{=} &\{\mathsf{Fr} \mapsto \mathsf{U}\,(\mathsf{PS}_1, \mathsf{PS}_2) \mid \mathsf{Fr} \mapsto \mathsf{PS}_1 \in \Psi^{\mathbf{fr}}_1, \mathsf{Fr} \mapsto \mathsf{PS}_2 \in \Psi^{\mathbf{fr}}_2\}\\ &\cup \{\mathsf{Fr} \mapsto \mathsf{PS}_1 \in \Psi^{\mathbf{fr}}_1 \mid \mathsf{Fr} \mapsto \mathsf{PS}_2 \notin \Psi^{\mathbf{fr}}_2\}\\ &\cup \{\mathsf{Fr} \mapsto \mathsf{PS}_2 \in \Psi^{\mathbf{fr}}_2 \mid \mathsf{Fr} \mapsto \mathsf{PS}_1 \notin \Psi^{\mathbf{fr}}_1\}\end{aligned}$$

$$\begin{aligned}\textsc{Join}(\mathsf{op}, (\Delta \rightsquigarrow v), \Psi^{\mathbf{fr}}_1, \Psi^{\mathbf{fr}}_2) \stackrel{\text{def}}{=} \textsc{Merge}(\{&((\Delta \rightsquigarrow v) \to \mathsf{Asm}_1 \sqcup \mathsf{Asm}_2) \mapsto \bowtie_{\mathsf{op}} (\mathsf{PS}_1, \mathsf{PS}_2) \mid\\ &(((\Delta_j \rightsquigarrow v_j) \to \mathsf{Asm}_j) \mapsto \mathsf{PS}_j) \in \Psi^{\mathbf{fr}}_j, j \in \{1, 2\}\})\end{aligned}$$
$$\text{where } \Delta_1 \rightsquigarrow v_1 \text{ and } \Delta_2 \rightsquigarrow v_2 \text{ are sufficient specification of } \Delta \rightsquigarrow v$$

Fig. 5. Operations that compose indexed extended version spaces.

concatenating traces. The consistency of the result follows from the observational soundness of VS intersection.

- *Cons operation.* The Cons operation takes a consistent frame-VS mapping $\mathsf{Fr} \mapsto \mathsf{PS}$, and a consistent traces-indexed VS $\Psi^{\mathbf{trs}}$. Then, it constructs a trace-indexed VS by Consing Fr onto Trs, the indices of $\Psi^{\mathbf{trs}}$, and taking the intersection of two VSes consistent with Fr and Trs. The consistency of the resulting $\Psi^{\mathbf{trs}}$ follows from Rule TERec.
- *Merge operation.* The Merge operation takes two consistent frame-indexed VSes $\Psi^{\mathbf{fr}}_1$ and $\Psi^{\mathbf{fr}}_2$, and merges them by taking the union of PS_1 and PS_2 consistent with the same Fr. The consistency of the resulting $\Psi^{\mathbf{fr}}$ follows from the semantics of $\mathsf{U}\,(\cdot)$.
- *Join operation.* Like join nodes in VSes, the Join operation is parametrized by an n-arity operator op. It also takes a synthesis problem specified by $\Delta \rightsquigarrow v$ along with n consistent frame-indexed VSes $\Psi^{\mathbf{fr}}_j$, denoting the solutions to synthesis sub-problems $\Delta_j \rightsquigarrow v_j$ for op's arguments. Then it reconstructs a frame-indexed VS for $\Delta \rightsquigarrow v$ by concatenating assumptions in $\Psi^{\mathbf{fr}}_j$ while joining the corresponding VSes with $\bowtie_{\mathsf{op}} (\cdot)$, whose consistency follows from Rules AERec, AECtor, and AEMatch.

5 SYNTHESIS ALGORITHM

In this section, we present our instantiation of the top-level synthesis procedure from Algo. 1. We first define our layering $(\mathcal{L}_j)_{j \in \mathbb{N}}$ of the search space $\mathcal{L}$. We then present a trace-guided Learn procedure to learn a traces-indexed VS from the user-provided I/O examples over a finite subspace $\mathcal{L}_j$ of this layered search space. We defer to Sec. 6 the specifics of our trace-based ranking function over traces-indexed VSes which is used to select the program returned by Algo. 1. The discussion focuses on the body e of the program `fix` $\mathsf{f}(x_{\mathsf{in}}).e$ to be synthesized.

$$\frac{\mathsf{isAtomic}(e) \quad \mathsf{ASTSize}(e) \leq \mu(\mathsf{atomicSize})}{e \vdash \mu}\ \textsc{SAtomic} \qquad \frac{\mu(\mathsf{ctorDepth}) > 0 \quad \mu' = \mu[\mathsf{ctorDepth}{--}] \quad e' \vdash \mu'}{C(e') \vdash \mu}\ \textsc{SCtor}$$

$$\frac{\mu(\mathsf{matchDepth}) > 0 \qquad \mu_{base} = \mu[\mathsf{matchDepth}{--}] \qquad \mu_{rec} = \mu_{base}[\mathsf{ctorDepth}{++}] \qquad \forall \mathrm{j}.\ arity(C_\mathrm{j}) = 0 \Rightarrow e_\mathrm{j} \vdash \mu_{base} \qquad \forall \mathrm{j}.\ arity(C_\mathrm{j}) > 0 \Rightarrow e_\mathrm{j} \vdash \mu_{rec}}{\texttt{match}\ x\ \texttt{with}\ \{\ldots C_\mathrm{j}(x_\mathrm{j}) \rightarrow e_\mathrm{j} \ldots\} \vdash \mu}\ \textsc{SMatch}$$

$$\frac{\mu(\mathsf{condWidth}) \geq n \qquad \forall \mathrm{j}.\ \mathsf{isAtomic}(g_\mathrm{j}) \qquad \forall \mathrm{j}.\ \mathsf{ASTSize}(g_\mathrm{j}) \leq \mu(\mathsf{atomicSize}) \qquad \forall \mathrm{j}.\ e_\mathrm{j} \vdash \mu}{\texttt{cond}\ \{\ldots g_\mathrm{j} \rightarrow e_\mathrm{j} \ldots\} \vdash \mu}\ \textsc{SCond}$$

Fig. 6. Satisfaction of a program P (its function body) with respect to a μ-value.

5.1 Layered Search Space

In contrast to commonly used size-based search space layering, we layer our search space $\mathcal{L}$ in a more *artful* way to assimilate domain-specific *structural attributes* or *shapes* of programs. Specifically, we provide (1) a tuple μ of parameters of a program's AST, (2) a characterization of the finite subspace $\mathcal{L}_\mu$ of $\mathcal{L}$ defined by each μ-value, and (3) an ordering over μ-values that yields a sequence $(\mathcal{L}_{\mu_\mathrm{j}})_{\mathrm{j}\in\mathbb{N}}$. We describe the parametrization μ in Fig. 6 by defining judgment $\mathsf{P} \vdash \mu$, denoting that program P satisfies the tuple μ of parameter values. Then the subspace $\mathcal{L}_\mu$ is characterized by the μ-value in the sense that $\mathsf{P} \in \mathcal{L}_\mu$ iff $\mathsf{P} \in \mathcal{L}$ and $\mathsf{P} \vdash \mu$. The specific ordering strategy used by us is a low-level heuristic and is hence deferred to Sec. 6.

Now we motivate each parameter in μ, and describe judgment $\mathsf{P} \vdash \mu$. (1) matchDepth specifies the maximum depth of pattern matching (Rule SMatch), and thus helps bias against consuming too much data from input at a time. (2) ctorDepth specifies the maximum depth of constructors (Rule SCtor) without pattern matching, and is set to be 0 by default. As we want to favor the usage of constructor when data is consumed from input via pattern matching,[3] the third premise in Rule SMatch increments ctorDepth for `match` branches with patterns of non-zero arity. (3) atomicSize specifies the maximum size of atomic expressions (IsAtomic), namely function applications, constructors, variables, and their compounds, and bounds the size of enumeration in effect. (4) condWidth specifies the maximum width of `cond` expressions (Rule SCond), and thus limits the complexity of branching logic in the programs.

Example 5.1. Consider the following two semantically equivalent implementations of the Fibonacci number function: (1) $e^* \equiv \texttt{cond}\ \{x_\mathsf{in} = 0 \rightarrow 1, x_\mathsf{in} = 1 \rightarrow 1, x_\mathsf{in} > 1 \rightarrow \mathsf{f}\,((x_\mathsf{in} - 2)) + \mathsf{f}\,((x_\mathsf{in} - 1))\}$; (2) $e^\dagger \equiv \texttt{match}\ x_\mathsf{in}\ \texttt{with}\ \{\mathsf{zero} \rightarrow 1, \mathsf{succ}(n) \rightarrow \texttt{match}\ n\ \texttt{with}\ \{\mathsf{zero} \rightarrow 1, \mathsf{succ}(\mathtt{m}) \rightarrow \mathsf{f}\,(\mathtt{m}) + \mathsf{f}\,(\mathtt{n})\}\}$. Subspace $\mathcal{L}_\mu$ contains e^* when $\mu(\mathsf{atomicSize})$ is at least 9 and $\mu(\mathsf{condWidth})$ is at least 3, and contains $e^\dagger$ when $\mu(\mathsf{atomicSize})$ is at least 5 and $\mu(\mathsf{matchDepth})$ is at least 2.

5.2 Trace-Guided Synthesis

We present a trace-guided synthesis procedure Learn (Algo. 2) which iterates through the set $\mathcal{E}$ of user-provided I/O examples to generate a traces-indexed VS $\Psi^{\mathbf{trs}}$ of programs in subspace $\mathcal{L}_\mu$, along with their recursion traces, that is consistent with $\mathcal{E}$. [4] This traces-indexed VS is constructed by iteratively computing the product (Prod, Line 9) over the trace-indexed VS $\Psi^{\mathbf{tr}}$ of programs consistent with each I/O example $\mathrm{i} \rightsquigarrow \mathrm{o}$.

[3] In contrast, "corecursive calls" may only take place under guarding constructors to ensure productivity [Coquand 1994].
[4] For brevity, we omit the synthesis context, which may include definition of background functions.

Algorithm 2 Trace-guided synthesis

Input: I/O examples $\mathcal{E}$
Input: Search space $\mathcal{L}_\mu$
Output: Traces-indexed VS $\Psi^{\mathbf{trs}}$

1: **procedure** LEARN($\mathcal{E}, \mathcal{L}_\mu$)
2: $\quad\Psi^{\mathbf{trs}} \leftarrow \{[] \mapsto \top\}$
3: $\quad$**for all** $\mathsf{i}\rightsquigarrow\mathsf{o} \in \mathcal{E}$ **do**
4: $\quad\quad\Psi^{\mathbf{tr}} \leftarrow \emptyset$
5: $\quad\quad\Psi^{\mathbf{fr}} \leftarrow$ LEARNANGELIC($\mathsf{i}\rightsquigarrow\mathsf{o}, \mathcal{L}_\mu$)
6: $\quad\quad$**for all** $(\mathsf{Fr} \mapsto \mathsf{PS}) \in \Psi^{\mathbf{fr}}$ **do** ▷ $\mathsf{Fr} = (\mathsf{i}\rightsquigarrow\mathsf{o})\rightarrow\mathsf{Asm}$ for some Asm
7: $\quad\quad\quad\Psi^{\mathbf{tr}}_{temp} \leftarrow$ CONS $\left(\mathsf{Fr} \mapsto \mathsf{PS},\ \text{LEARN}(\mathsf{Asm}, \mathcal{L}_\mu)\right)$
8: $\quad\quad\quad\Psi^{\mathbf{tr}} \leftarrow$ CONCAT($\Psi^{\mathbf{tr}}, \Psi^{\mathbf{tr}}_{temp}$)
9: $\quad\quad\Psi^{\mathbf{trs}} \leftarrow$ PROD($\Psi^{\mathbf{trs}}, \Psi^{\mathbf{tr}}$)
10: $\quad$**return** $\Psi^{\mathbf{trs}}$

The interesting part of LEARN then lies in Lines 4—8, where LEARN constructs $\Psi^{\mathbf{tr}}$ for I/O example $\mathsf{i}\rightsquigarrow\mathsf{o}$. [5] First, LEARN uses a sub-procedure LEARNANGELIC (Algo. 3) to collect angelic solutions, along with the corresponding execution frames, in a frame-indexed VS $\Psi^{\mathbf{fr}}$. Then, for each frame-VS mapping $\mathsf{Fr} \mapsto \mathsf{PS}$ in $\Psi^{\mathbf{fr}}$, LEARN recursively calls itself with the assumptions Asm from Fr as the new synthesis task (Line 7), and uses the output traces-indexed VS of this recursive call and the frame-VS mapping to iteratively build up $\Psi^{\mathbf{tr}}$ (Line 7 and Line 8). Recall that the CONS operation on Line 7 not only computes an intersection between the expressions in PS and in the output of LEARN, but also splices together Fr and the traces Trs in the output of LEARN into a new trace. The CONCAT operation on Line 8 collects the trace-indexed VSes for each mapping into one trace-indexed VS for the current I/O example.

Example 5.2. The user provides an example $\mathsf{Ex} = \mathtt{cons}(1, \mathtt{cons}(2, \mathtt{cons}(3, \mathtt{nil})))\rightsquigarrow 6$, and intends to have the synthesizer find a program that produces the sum of elements in the input list. LEARN calls LEARNANGELIC on Ex , and has it return a frame-indexed VS $\Psi^{\mathbf{fr}} = \{\ldots, (\mathsf{Ex}\rightarrow\mathsf{Asm}) \mapsto \mathsf{PS}, \ldots\}$. Say $\mathsf{Asm} = [\mathtt{cons}(2, \mathtt{cons}(3, \mathtt{nil}))\rightsquigarrow 5]$ and PS contains $e^* \equiv \mathtt{match}\ x_{\mathsf{in}}\ \mathtt{with}\ \{\mathtt{nil} \rightarrow 0, \mathtt{cons}(\mathtt{hd}, \mathtt{tl}) \rightarrow \mathtt{hd}+\mathtt{f}(\mathtt{tl})\}$. Recursively calling LEARN on Asm may produce a traces-indexed VS $\Psi^{\mathbf{trs}} = \{\ldots, [\mathsf{Tr}] \mapsto \mathsf{PS}', \ldots\}$ where Tr is a linear trace $(\mathtt{cons}(2, \mathtt{cons}(3, \mathtt{nil}))\rightsquigarrow 5)\rightarrow[(\mathtt{cons}(3, \mathtt{nil})\rightsquigarrow 3)\rightarrow[(\mathtt{nil}\rightsquigarrow 0)\rightarrow[]]]$ and PS′ also contains e^*.
As a result, the traces-indexed VS produced by calling LEARN on the user-provided Ex is of the form $\{\ldots, [\mathsf{Ex}\rightarrow[\mathsf{Tr}]] \mapsto \mathsf{PS} \cap \mathsf{PS}', \ldots\}$. And the desired implementation e^*, of course, is contained in $\mathsf{PS} \cap \mathsf{PS}'$. □

5.3 Angelic Synthesis

Given a synthesis problem specified by an example $\Delta\rightsquigarrow v$ and a bounded search space $\mathcal{L}_\mu$, the synthesis procedure LEARNANGELIC (Algo. 3) constructs a frame-indexed VS $\Psi^{\mathbf{fr}}$ of expressions in $\mathcal{L}_\mu$, along with the assumptions that make them satisfy $\Delta\rightsquigarrow v$. The procedure is based on top-down enumerative search and lightweight deductive reasoning. In particular, the procedure uses deduction for expressions with top-level operators that enable deductive reasoning (Lines 3—5) and uses enumeration for atomic expressions (Line 6). At each step, the MERGE operation is used to

[5] In our implementation, trace-indexed VSes are memoized.

Algorithm 3 Angelic synthesis

Input: Valuation context Δ
Input: Goal value v
Input: Search space $\mathcal{L}_\mu$
Output: Frame-indexed VS Ψ^{fr}

1: **procedure** LearnAngelic($\Delta \rightsquigarrow v, \mathcal{L}_\mu$)
2: $\quad\Psi^{\text{fr}} \leftarrow \emptyset$
3: $\quad$**for all** op $\in$ DeducibleOp($\mathcal{L}_\mu$) **do**
4: $\quad\quad\Psi^{\text{fr}}_{temp} \leftarrow$ LearnOp(op, $\Delta \rightsquigarrow v$, $\mathcal{L}_\mu$)
5: $\quad\quad\Psi^{\text{fr}} \leftarrow$ Merge(Ψ^{fr}, Ψ^{fr}_{temp})
6: $\quad\Psi^{\text{fr}}_{enum} \leftarrow$ LearnEnum($\Delta \rightsquigarrow v, \mathcal{L}_\mu$)
7: $\quad$**return** Merge($\Psi^{\text{fr}}, \Psi^{\text{fr}}_{enum}$)
8: **procedure** LearnOp(op, $\Delta \rightsquigarrow v$, $\mathcal{L}_\mu$)
9: $\quad$**for** j $\leftarrow 1, \ldots,$ Arity(op) **do**
10: $\quad\quad(\Delta_j \rightsquigarrow v_j) \leftarrow \text{Deduce}_{\text{op,j}}(\Delta \rightsquigarrow v)$
11: $\quad\quad\mathcal{L}_{\mu_j} \leftarrow \text{WalkDown}_{\text{op,j}}(\mathcal{L}_\mu)$
12: $\quad\quad\Psi^{\text{fr}}_j \leftarrow$ LearnAngelic($\Delta_j \rightsquigarrow v_j, \mathcal{L}_{\mu_j}$)
13: $\quad$**return** Join(op, $\Delta \rightsquigarrow v, \ldots, \Psi^{\text{fr}}_j, \ldots$)
14: **procedure** LearnEnum($\Delta \rightsquigarrow v, \mathcal{L}_\mu$)
15: $\quad\Psi^{\text{fr}} \leftarrow \emptyset$
16: $\quad$**for all** $e \in$ AtomicExp($\mathcal{L}_\mu$) **do**
17: $\quad\quad$**for all** Asm$\in$InferAsm($[\Delta]\, e, v$) **do**
18: $\quad\quad\quad\Psi^{\text{fr}} \leftarrow$ Merge(Ψ^{fr},
19: $\quad\quad\quad\quad\{((\Delta \rightsquigarrow v) \rightarrow \text{Asm}) \mapsto \{e\}\}$)
20: $\quad$**return** Ψ^{fr}

combine frame-indexed VSes by grouping the corresponding angelic solutions by their assumptions (Line 5, Line 6).

The procedure LearnOp implements a general top-down algorithm that propagates the angelic synthesis problem, specified by example $\Delta \rightsquigarrow v$ and search space $\mathcal{L}_\mu$, into the arguments of an operator op in tinyML. The Join operation then combines the angelic solutions for each argument by joining the respective version spaces and concatenating the respective assumptions.

The only two "deducible" operators in tinyML are $C(\cdot)$ and `match` x `with` $\{\ldots C_j(x_j) \rightarrow \cdot \ldots\}$. And thus, we present Deduce procedures for them as follows. Notice that, in both cases, at most one example per argument is deduced. In particular, only one branch of `match` will get an example[6].

$$\text{Deduce}_{\texttt{Ctor},j}(\Delta \rightsquigarrow v) = \Delta \rightsquigarrow v_j \quad \text{for all j, where } v = C(\ldots, v_j, \ldots)$$

$$\text{Deduce}_{\texttt{match}(x),j}(\Delta \rightsquigarrow v) = \begin{cases} (\Delta, x_j := v') \rightsquigarrow v & \text{if } [\Delta]\, x = C_j(v') \\ \emptyset & \text{otherwise} \end{cases}$$

The procedure LearnEnum iterates through all atomic expressions e such that $\text{ASTSize}(e) \leq \mu(\text{atomicSize})$, and collects, in a frame-indexed VS, assumptions that make e angelically evaluate to the goal value v.

The procedure InferAsm is responsible for identifying all assumptions Asm such that $e \Downarrow^{\text{Asm}}_{[]} v$ (Fig. 4). The key idea in the procedure is to identify the desired outputs for recursive subcalls in e that make e evaluate to v using an SMT solver (Z3 in our case). First, we substitute Δ into e so that f is the only free variable in e if any; for variables denoting background functions, we substitute them with the Z3 encoding of their functional definitions. Then we substitute all recursive subcalls $\text{f}\,(\text{i}_j)$ in e with a fresh symbol α_{i_j} denoting their yet-to-be-determined outputs (making sure that the same symbol is used for recursive subcalls sharing the argument i_j). [7] This leaves us a formula ϵ such that $e \equiv \left[\alpha_{\text{i}_j} := \text{f}\,(\text{i}_j)\right] \epsilon$. Repeatedly, we query Z3 for satisfying assignments $\alpha_{\text{i}_j} \hookrightarrow \text{o}_j$ of formula $\phi \equiv (\epsilon = v)$, and add conjuncts, that negate the seen assignments, to ϕ until ϕ becomes unsatisfiable. Each set of satisfying assignments $\alpha_{\text{i}_j} \hookrightarrow \text{o}_j$ becomes a set of consistent assumptions about f, i.e., $\text{f}(\text{i}_j) = \text{o}_j$.

[6]The other branches get no example, and we use a placeholder $\top$ for them to indicate that any expression of type τ works.
[7]i_j has to be a value rather than a nested recursive subcalls since we only permit structural recursion.

Example 5.3. Recall in Example 5.2, LEARN passes an example $\texttt{cons}(1, \texttt{cons}(2, \texttt{cons}(3, \texttt{nil}))) \rightsquigarrow 6$ to LEARNANGELIC. Calling LEARNANGELIC on this Ex may produce a frame-indexed VS $\Psi^{\mathbf{fr}} = \{(\mathsf{Ex} \rightarrow \mathsf{Asm}_1) \mapsto \mathsf{PS}_1, (\mathsf{Ex} \rightarrow \mathsf{Asm}_2) \mapsto \mathsf{PS}_2, \ldots\}$ where:

- $\mathsf{Asm}_1 = [\texttt{cons}(2, \texttt{cons}(3, \texttt{nil})) \rightsquigarrow 5]$ and PS_1 contains $\texttt{match}\ x_{\mathsf{in}}\ \texttt{with}\ \{\texttt{nil} \rightarrow 0, \texttt{cons}(\texttt{hd}, \texttt{tl}) \rightarrow \texttt{hd} + \texttt{f}(\texttt{tl})\}$, and
- $\mathsf{Asm}_2 = [\texttt{cons}(2, \texttt{cons}(3, \texttt{nil})) \rightsquigarrow 6]$ and PS_2 contains $\texttt{match}\ x_{\mathsf{in}}\ \texttt{with}\ \{\texttt{nil} \rightarrow 1, \texttt{cons}(\texttt{hd}, \texttt{tl}) \rightarrow \texttt{hd} \times \texttt{f}(\texttt{tl})\}$.

In the first case, assuming $\texttt{f}([2,3]) = 5$, $\texttt{head}(x_{\mathsf{in}}) + \texttt{f}(\texttt{tail}(x_{\mathsf{in}}))$ conforms to Ex and similarly, assuming $\texttt{f}([2,3]) = 5$, $\texttt{head}(x_{\mathsf{in}}) \times \texttt{f}(\texttt{tail}(x_{\mathsf{in}}))$ conforms to Ex. Note that, though the two programs mentioned here happen to truly satisfy Ex, PS_1 and PS_2 may contain programs that only *angelically* satisfy Ex. $\texttt{match}\ x_{\mathsf{in}}\ \texttt{with}\ \{\texttt{nil} \rightarrow 1, \texttt{cons}(\texttt{hd}, \texttt{tl}) \rightarrow \texttt{hd} + \texttt{f}(\texttt{tl})\}$ is such an angelic candidate program that will be pruned away during VS intersection on Line 7 of Algo. 2. □

5.4 Soundness and Completeness

Since INFERASM returns all assumptions Asm such that $e \Downarrow_{[]}^{\mathsf{Asm}} v$, and operations MERGE and JOIN are consistency preserving (as discussed in Sec. 4.3), frame-indexed VSes returned by LEARNANGELIC are consistent by construction. As a result, LEARNANGELIC is sound (Thm. 5.4).

Theorem 5.4 (Soundness of LEARNANGELIC)**.** For all $\mathsf{Fr} \mapsto \mathsf{PS} \in \textsc{LearnAngelic}(\mathsf{Ex}, \mathcal{L}_\mu)$ and for all $e \in \mathsf{PS}$, we have $e \in \mathcal{L}_\mu$ and $[\Delta]\, e \Downarrow_{[]}^{\mathsf{Asm}} v$ where $\mathsf{Fr} = (\Delta \rightsquigarrow v) \rightarrow \mathsf{Asm}$.

Since operations CONCAT, PROD, and CONS are consistency preserving (as discussed in Sec. 4.3), the soundness of LEARN (Thm. 5.5) follows directly from Thm. 5.4.

Theorem 5.5 (Soundness of LEARN)**.** For all $\mathsf{Trs} \mapsto \mathsf{PS} \in \textsc{Learn}(\mathcal{E}, \mathcal{L}_\mu)$ and for all $e \in \mathsf{PS}$, we have that $e \in \mathcal{L}_\mu$ and $(\texttt{fix}\ \texttt{f}(x).e) \models \mathsf{Trs}$.

Now, we discuss the completeness of procedures LEARNANGELIC and LEARN. Careful readers may have noticed that neither procedure LEARNDEDUCE nor procedure LEARNENUM considers `cond` expressions. This is because neither deduction nor enumeration scales well for synthesis of `cond` expressions. The return value of LEARNANGELIC, *implicitly* and thus efficiently, carries `cond` expressions by only storing their branches, which may later be unified into choice nodes during VS intersection (Line 7 and Line 9)!

As a result, the completeness of LEARN and LEARNANGELIC holds modulo conditional expressions. That is, when an expression e uses a conditional expression when it is not necessary, it is not guaranteed that it is present in the return value of LEARNANGELIC or LEARN — only that another expression e' (possibly conditional free) that is observationally equivalent to e is present.

Theorem 5.6 (Observational Completeness of LEARNANGELIC)**.** If $e \in \mathcal{L}_\mu$ and $[\Delta]\, e \Downarrow_{[]}^{\mathsf{Asm}} v$, then there exists $((\Delta \rightsquigarrow v) \rightarrow \mathsf{Asm}) \mapsto \mathsf{PS} \in \textsc{LearnAngelic}(\Delta \rightsquigarrow v, \mathcal{L}_\mu)$ and $e' \in \mathsf{PS}$ such that $e' \equiv_{\mathsf{Support}(\mathsf{PS})} e$.

Theorem 5.7 (Observational Completeness of LEARN)**.** If $e \in \mathcal{L}_\mu$ and $(\texttt{fix}\ \texttt{f}(x).e) \models \mathsf{Trs}$, then there exists $\mathsf{Trs} \mapsto \mathsf{PS} \in \textsc{Learn}(\mathsf{Roots}(\mathsf{Trs}), \mathcal{L}_\mu)$ and $e' \in \mathsf{PS}$ such that $e' \equiv_{\mathsf{Support}(\mathsf{PS})} e$.

6 IMPLEMENTATION

We implement a type-directed synthesizer, called SYRUP, in OCaml from scratch. Instead of parsing user-provided definitions, SYRUP incorporates inbuilt definitions of algebraic data types and background functions along with their Z3 encodings. In this section, we will expand on important implementation details and heuristics omitted in previous sections.

6.1 Parameterized Search Space Layering

In Sec. 5.1, we have seen that the search space parametrization enables us to concentrate on programs with a specific *shape* rather than a certain *size* in each synthesis iteration.

We now describe how SyRup changes parameters across synthesis iterations, thereby prioritizing the search for programs of certain shapes over others. Parameters atomicSize, matchDepth, and condWidth are successively increased in lexicographic order; first atomicSize is incremented up to an upper bound; if a solution isn't found, then matchDepth is incremented, while atomicSize is reset to its initial value and incremented again upto an upper bound; and so on. Similar methods have been used in prior work [Lubin et al. 2020], and are referred to as "staging".

Finally, we use initial values of the parameters that are sufficiently large to encompass interesting programs in the search space, but not excessively large to prevent the inclusion of improbable programs in the search space. Specifically, we initially let atomicSize = 3 to include function application, matchDepth = 1 since pattern matching is ubiquitous in functional programming, and condWidth = 1 to avoid overfitting.

6.2 Program Ranking Function

Program rank plays an important role in picking programs that generalize to unseen examples as we will see in Sec. 7.4. From the traces-indexed VS $\Psi^{\mathbf{trs}}$ returned by Algo. 2, SyRup picks Trs $\mapsto$ PS that maximizes the number of unique assumptions in Trs, favoring candidate programs that exploit more input information via recursion. Then, from PS, SyRup picks a program with the smallest AST depth.

In the presence of choice nodes in the VSes in $\Psi^{\mathbf{trs}}$, SyRup proceeds slightly differently. In particular, SyRup first attempts to select a program using the above ranking function without any match expressions in its conditional guards. If this fails, SyRup then attempts to select a program using the above ranking function with at most 1 match expression in its guards.

6.3 Tracking of Recursion Traces

Instead of tracking complete recursion traces, we choose to track an abstraction of recursion traces, specifically, the unique recursive sub-calls subsumed. Recall that we track supporting valuation contexts for our extended version spaces; the supports of top-level VS nodes exactly corresponds to the input of recursive sub-calls made by the programs contained. For example, a join node that contains the program in the first row of Fig. 1 has the following supports {[m := 2, n := 1], [m := 1, n := 1], [m := 0, n := 1]}. While this abstraction is sufficient for our benchmarks and evaluation, we defer the exploration of more precise abstractions to future work.

7 EVALUATION

In our experimental evaluation of SyRup's performance, we consider the following research questions.

RQ1 How does SyRup perform on various synthesis tasks compared to existing techniques?
RQ2 How sensitive is SyRup's performance to the quality of the I/O examples?
RQ3 How does SyRup's layered search space and ranking function impact its overall performance?

All experiments are performed in a single thread on a department-wide shared Linux server equipped with two 2.90GHz Intel Xeon E5-2690 2.90GHz 8-core processors and 192GB of RAM, with a time limit of 120 seconds.

7.1 Experimental Setup

Synthesis Tasks. A synthesis task consists of a programming task T (e.g., write a program that adds two natural numbers) and a set of I/O examples e.g. $\{(1,1)\rightsquigarrow 2, (2,1)\rightsquigarrow 3\}$). We use a suite of 43 functional programming tasks that have been used to evaluate prior work [Lubin et al. 2020; Osera and Zdancewic 2015]. Of these, 32 tasks require recursion. For each programming task T, we spawn numerous synthesis tasks by generating I/O examples using the following two strategies. These strategies simulate the expertise of different users, allowing us to assess the performance of SyRup in diverse scenarios.

(1) **Expert Examples**: Each programming task in our suite was originally developed with a corresponding *trace-complete* set of I/O examples [Osera and Zdancewic 2015]. Similar to evaluations in prior work (Smyth, Burst), we evaluate our synthesizer's performance on subsets of these trace-complete example sets. Because such subsets are aware of the (recursive) semantics of the desired program to some degree, we regard them as *expert examples*. However, evaluations in prior work only use a *single, manually crafted subset* of the original trace-complete example set for each programming task. To effectively examine how well a synthesizer generalizes to unseen inputs given expert examples, we argue that it is important to evaluate across different subsets of these trace-complete example sets. In our experiments, we use subsets of different sizes, ranging from 1 to the total number of available expert examples, sampling up to 10 subsets for each size.

(2) **Random Examples**: Expert examples, while not necessarily trace-complete, serve as a reasonable approximation of a user who is familiar with recursive programs and synthesis tools. To evaluate SyRup's performance on examples provided by non-experts, we generate random inputs and execute reference implementations on them, obtaining 10 example sets of sizes ranging from 1 to 6. To ensure that these example sets can be provided by a human user, we bound the number of constructors allowed in the input values, following a similar approach to Lubin et al. [2020]. For inputs with a function type, we randomly select from the functions used in expert examples.

Success Rate. To evaluate a synthesizer's performance on a programming task T under various scenarios, we spawn synthesis tasks associated with T using I/O example sets with different attributes (identified by example generation strategies and number of examples). Then for a set $\mathbb{E}_\mathsf{T}$ of example sets $\mathcal{E}$ with the same attribute, we compute the synthesizer's *success rate* $S_{\mathbb{E}_\mathsf{T}}$ as follows:

$$S_{\mathbb{E}_\mathsf{T}} = \frac{|\{\mathcal{E} \in \mathbb{E}_\mathsf{T}| \text{ given } \mathcal{E}, \text{ the synthesizer generates a correct program for T }\}|}{|\mathbb{E}_\mathsf{T}|}$$

Baselines. We compare SyRup against the two state-of-the-art tools, Smyth [Lubin et al. 2020] and Burst [Miltner et al. 2022], for synthesis of recursive functional programs from examples. Neither Smyth nor Burst requires trace-complete specifications. While Smyth is based on top-down propagation and partial evaluation, Burst performs bottom-up search using a tree automata-based program representation.

We do not consider other synthesizers for recursive functional programs such as Leon [Kneuss et al. 2013], λ^2[Feser et al. 2015], and Synquid [Polikarpova et al. 2016] in our evaluation. Both Leon and Synquid are less effective in handling example-based specifications, and, in particular, ones that are not trace-complete [Lubin et al. 2020]. While λ^2*can* handle examples that are not trace-complete, the technique heavily relies on the use of language-level recursion schemes in the form of higher-order combinators such as `map` and `fold`.

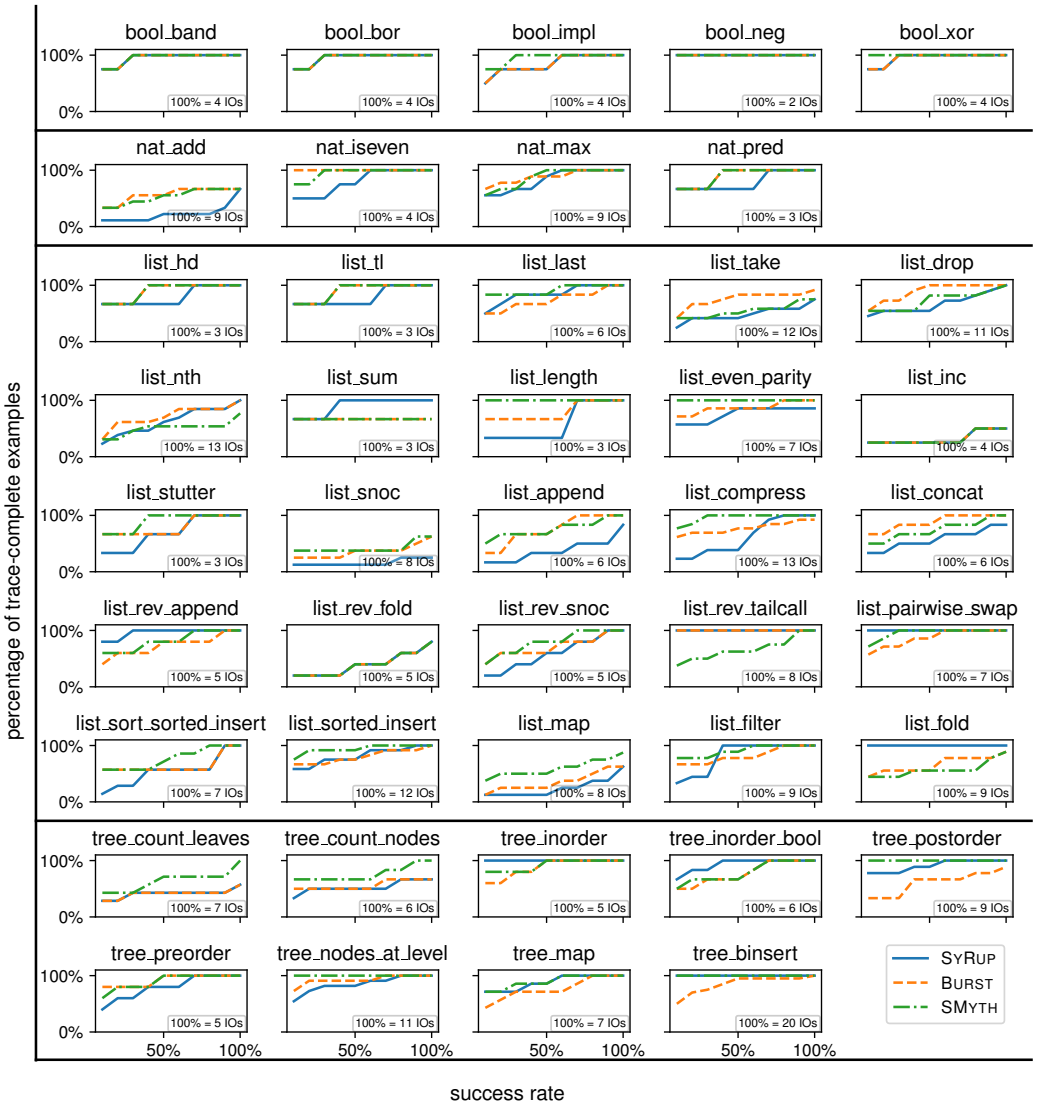

Fig. 7. Percentage of trace-complete examples needed to achieve specific success rates for each programming task. The x-axis label in each sub-figure indicates the size of the trace-complete example set for the corresponding programming task.

7.2 SyRup's Performance on Expert Examples

To evaluate SyRup's performance in the absence of trace-complete examples, we spawn synthesis tasks using sets of expert examples of different sizes for each programming task, and compute the respective success rates as discussed earlier. In Fig. 7, we present the *percentages of trace-complete examples* needed to achieve specific success rates (ranging from 10%, 20%, . . . , 100%) for each programming task [8].

[8] `tree_inorder_bool` corresponds to `tree_collect_leaves` in prior work. We correct the name to reflect its true semantics.

For all 43 programming tasks, SyRup overall needs less or similar number of examples to achieve the same success rate. For instance, to achieve 50% success rate, SyRup needs less examples on 16 tasks than both Burst and Smyth, and only needs more examples on 5 tasks than both of them (`list_sum`, `list_rev_append`, `list_filter`, `list_fold`, and `tree_inorder_bool`). This is better demonstrated in Fig. 8, which shows the percentage of trace-complete examples needed on average to achieve specific success rates. For instance, to achieve 50% success rate, SyRup needs 70%, Burst needs 76%, and Smyth needs 81% of trace-complete examples on average.

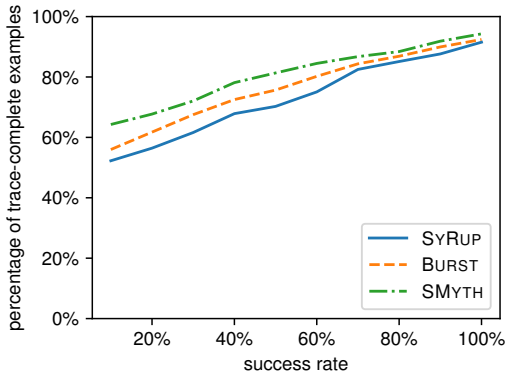

Fig. 8. Average percentage of trace-complete examples needed to achieve specific success rates.

7.3 SyRup's Sensitivity to the Quality of Examples

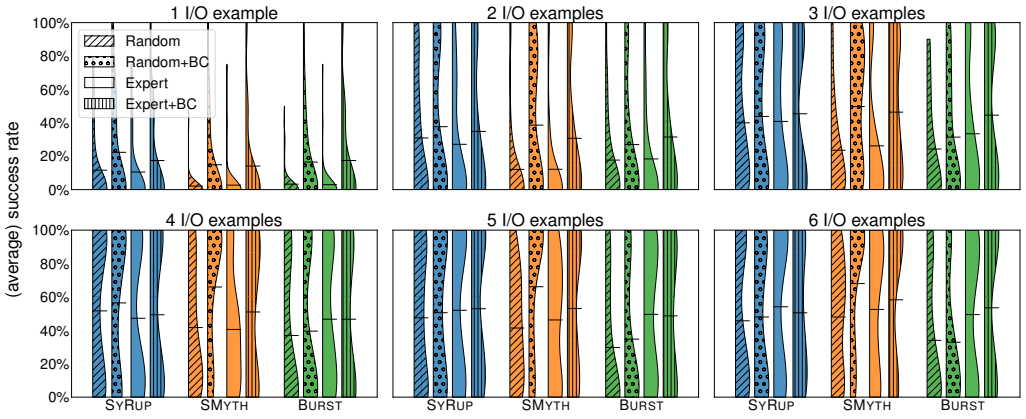

Fig. 9. Average success rate on example sets of size 1 to 6.

To assess the sensitivity of SyRup to the quality of examples, we spawn synthesis tasks on example sets generated in different ways. Then, we present the average (the horizontal bars) and the variation (the violin plots) of success rate across all 43 programming tasks in Fig. 9. In addition to the two strategies for example generation mentioned already, we differentiate between example sets that *must* or *may* contain the *base case* for the recursive programming tasks (i.e. the example(s) with the smallest input). This yields 4 classes of example sets as shown in the figure (Random, Random + BC, Expert, Expert + BC). We repeat the experiment for example sets of sizes ranging from 1 to 6.

We observe that: (1) the average success rate of SyRup across all programming tasks exhibits less variation compared to Burst and Smyth when provided with the same number of I/O examples with different attributes; (2) the distribution of success rates of SyRup across all programming tasks demonstrates a consistent trend regardless of the attribute of the given I/O examples (the shapes of the violin plots within each subplot are similar for SyRup); and (3) with one exception

(discussed in Sec. 7.5), SyRup overall has a higher or comparable success rate. Additionally, the specific attributes of example sets used highlights that: (1) SyRup gains little improvement in success rate from the inclusion of base case examples, and thus does not fully rely on knowledge about the base case to synthesize the desired program; (2) providing random examples or expert examples has little impact on SyRup's success rate, and thus SyRup does not rely on users' mental model of the implementation to synthesize the desired program.

7.4 Ablation Study

Recall Algo. 1. To evaluate the impact of SyRup's design choice of search space layering and ranking function on its overall performance, we perform an ablation study using commonly-used variants for each design choice.

Layered Search Space. First, we implement a version of SyRup, called SyRup$^+$ that layers the search space based on AST height (similar to Burst) and rerun our experiment for expert examples. Unfortunately, SyRup$^+$ times out very often for 33 out of 43 programming tasks.

Trace-based Ranking Function. We next implement a version of SyRup, called SyRup*, that uses the same layered search space as SyRup but uses a different ranking function.

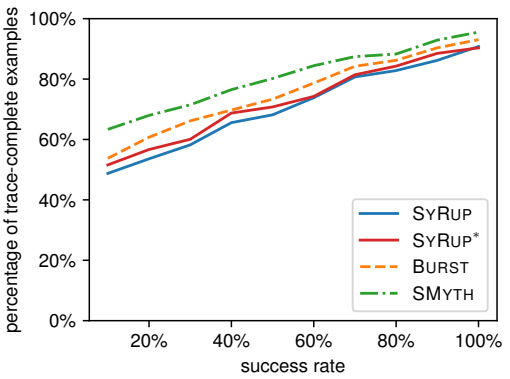

Fig. 10. Average percentage of trace-complete examples needed to achieve certain success rates across 32 recursive programming tasks.

In particular, SyRup* selects a program with the minimum AST height from the constructed traces-indexed VS. We rerun our experiment for expert examples. Since our ranking function greatly biases towards programs with longer recursion traces, the results shown in Fig. 10 focus on the 32 recursive tasks. As can be seen, both SyRup and SyRup* need less examples to achieve the same success rate on average than Burst and Smyth, with SyRup slightly outperforming SyRup*. After inspecting the performance difference on individual tasks manually (omitted in the paper to save space), we observe that for 8 out of 10 tasks on which SyRup outperforms (needs less examples to achieve the same success rate) SyRup*, SyRup also outperforms Burst and Smyth. Therefore, we conclude that, both trace-guided synthesis and trace-based ranking function contribute to SyRup's performance on expert examples.

7.5 Discussion

Evaluation Summary. Our evaluation demonstrates SyRup's ability to effectively tackle the challenges of generalization, trace-completeness, and ambiguity-resolution across diverse programming tasks and I/O example sets. In particular, the evaluation shows that SyRup requires fewer (trace-complete) examples than its counterparts to achieve the same success rates, *and* is significantly less sensitive to example quality. Thus, SyRup's trace-guided approach utilizing recursion traces as an inductive bias can benefit a variety of synthesis tasks for recursive programs.

Synthesis Time and Scalability. The synthesis time of SyRup is comparable to that of Smyth and Burst and is typically within 5 seconds or less when given a few examples with exceptions on a few relatively difficult tasks, which is often accompanied by time-outs. For instance, given 10 sets of expert examples of size 6, SyRup times out at least once on 6 tasks, and Burst times out at least once on 9 tasks, out of 43 programming tasks. On the other hand, Smyth, while rarely timing out, might fail to return a solution due to the absence of a base case example.

It is interesting to note that both SyRup and Burst start to time out more often when the number of examples increases or the quality of examples decreases (if we consider expert examples as high quality), while Smyth's runtime is less affected. However, this is expected as both SyRup and Burst predominantly rely on computationally expensive version space (tree automata for Burst) intersections. Consequently, we see in Sec. 7.3 that the increase in success rates of SyRup and Burst slows down as the number of user-provided examples increases, especially when the examples are randomly generated. While we plan to explore optimizations for version space intersection, it is helpful to note that scalability is not an issue for the current benchmarks and the intended use-case of SyRup only expects users to provide a small number of examples.

Failure Analysis and Limitations. SyRup fails to find a correct implementation for `list_rev_tailcall` because we only permit recursive subcalls on values that are strictly smaller than the input argument—this is also the case in prior work with the same assumption [Lubin et al. 2020; Miltner et al. 2022; Osera and Zdancewic 2015]. Furthermore, due to efficiency considerations, we limit scrutinees in `match` expressions to be variables. As a result, SyRup fails on `list_pairwise_swap`, whose correct implementation relies on *inside-out recursion* [Osera and Zdancewic 2015]. Allowing scrutinees to make recursive subcalls would require enumerating and checking the consistency of candidate programs as a whole since our algorithm relies on the concrete value of the scrutinee to deduce examples for branches in `match` expressions. Lastly, our implementation of the InferAsm decision procedure assumes that all background functions are first-order and monomorphic, as it is based on the Z3 Solver. While this limitation does not cause any issues with the current set of benchmarks, alternative or augmented implementations of InferAsm that overcome this limitation are worth exploring in future work.

8 RELATED WORK

Synthesis of Recursive Programs. Synthesis of recursive programs has witnessed great progress since Summers [1977], especially in the past decade. There are techniques that synthesize recursive programs from logical specifications [Itzhaky et al. 2021; Kneuss et al. 2013; Kuncak et al. 2010], refinement types [Polikarpova et al. 2016], or reference implementations [Farzan et al. 2022; Farzan and Nicolet 2021]. Here, we focus on synthesis of recursive programs from I/O examples. To enable efficient synthesis, initial attempts rely on either (1) comprehensive (trace-complete) examples, including Igor II [Hofmann 2010], Escher [Albarghouthi et al. 2013], and Myth [Osera and Zdancewic 2015], or (2) built-in recursion schemes, such as recursive higher-order functions used by λ^2 [Feser et al. 2015] and BIGλ [Smith and Albarghouthi 2016].

More recently, Smyth [Lubin et al. 2020] and Burst [Miltner et al. 2022] have relaxed the requirement. Specifically, Smyth devises a technique called *un-evaluation* and uses it in combination with best-effort forward evaluation [Omar et al. 2019] for top-down propagation of examples. Burst leverages angelic semantics to identify an over-approximating set of satisfying programs, picks the smallest program along with witnesses to the angelic satisfaction, and refines the program set with the witnesses, until a truly satisfying program is found. However, both Smyth and Burst can miss programs that generalize better, and, in particular, have more suitable recursion traces. SyRup extends the idea of angelic semantics to explore all possible traces in a bounded search space, and presents a novel VSA to represent and manipulate consistent trace-program pairs.

Representation-based Synthesis. The use of VSAs to learn and store sets of programs dates back to [Lau et al. 2003; Mitchell 1982]. Since FlashFill [Gulwani 2011] introduced a compact representation similar to e-graphs [Downey et al. 1980], multiple efforts have used similar representations in different problem settings [Barowy et al. 2015; Le and Gulwani 2014; Lee 2021; Polozov and Gulwani 2015; Rolim et al. 2017; Wang et al. 2016]. Another line of work uses finite tree automata (FTA) as a version space representation [Koppel et al. 2022; Miltner et al. 2022; Wang et al. 2017, 2018a,b] Our

work presents a richer representation augmented with recursion traces and conditional semantics. Generalization to other semantic features, compact representation of indexed extended version spaces, and adaption to FTAs, are all exciting opportunities for future work.

Condition Abduction/Unification. Conditional expressions have been widely used to improve the scalability of synthesis using a divide-and-conquer approach [Albarghouthi et al. 2013; Alur et al. 2017; Gulwani 2011; Ji et al. 2021; Lee 2021; Shi et al. 2019; Wang et al. 2017]. Most of these approaches introduce conditionals in an *eager* way. Leveraging the conditional semantics of extended version space, SyRup introduces conditionals in an *on-demand* (like Escher [Albarghouthi et al. 2013]), and *delayed* (like FrAngel [Shi et al. 2019]) manner. A potential direction for future work is extension of the conditional semantics of extended version spaces using unification mechanisms that go beyond conditionals [Alur et al. 2015].

Search Space Layering. Existing synthesizers typically use a bounded search space and increase the bound (e.g., on program ASTs or intermediate values) when the current search space is exhausted. While one can construct a larger search space from the smaller ones, the search space still grows exponentially with respect to the bound [Barke et al. 2020]. Inspired by Myth [Osera and Zdancewic 2015] and Smyth [Lubin et al. 2020], SyRup parameterizes functional programs in a domain-specific and structural way. The parameters allow us to layer the search space more finely so that the search space not only grows more slowly, but also focuses on programs of different shapes. This means that our technique is likely to benefit from parallelizing the search with different threads focusing on programs of different shapes like Bornholt et al. [2016]; Padhi et al. [2019].

Program Ranking. Existing synthesis techniques, especially those based on VSAs, use a monotonic function defined over program ASTs to pick a "smallest" program. Smaller programs may not always generalize better as we have seen in Fig. 1. To the best of our knowledge, the only work that considers semantic features for ranking programs is Ellis and Gulwani [2017]. However, their approach relies on post-synthesis computation to rank candidate programs. In contrast, we collect and compose semantic features *during* synthesis. Our notion of indexed extended version space also opens up opportunities for exploration of other semantic features with different levels of abstraction.

9 DATA-AVAILABILITY STATEMENT

Our implementation along with the code required for reproduction, is available at Yuan et al. [2023].

ACKNOWLEDGMENTS

We thank our anonymous reviewers and our shepherd, Hila Peleg, for their helpful feedback, and Patrick LaFontaine for his detailed comments on the initial draft of the paper. We thank Anders Miltner and Justin Lubin for answering our many questions about Burst and Smyth. This research was partially supported by the National Science Foundation under Grant No. 1846327. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, Berlin, Heidelberg, 934–950. https://doi.org/10.1007/978-3-642-39799-8_67

Rajeev Alur, Pavol Černý, and Arjun Radhakrishna. 2015. Synthesis Through Unification. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 163–179. https://doi.org/10.1007/978-3-319-21668-3_10

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *Tools and Algorithms for the Construction and Analysis of Systems*, Axel Legay and Tiziana Margaria (Eds.). Vol. 10205. Springer Berlin Heidelberg, Berlin, Heidelberg, 319–336. https://doi.org/10.1007/978-3-662-54577-5_18

Shraddha Barke, Hila Peleg, and Nadia Polikarpova. 2020. Just-in-Time Learning for Bottom-up Enumerative Synthesis. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 1–29. https://doi.org/10.1145/3428295

Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. 2015. FlashRelate: Extracting Relational Data from Semi-Structured Spreadsheets Using Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 218–228. https://doi.org/10.1145/2737924.2737952

Rastislav Bodik, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. 2010. Programming with Angelic Nondeterminism. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. Association for Computing Machinery, New York, NY, USA, 339–352. https://doi.org/10.1145/1706299.1706339

James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. 2016. Optimizing Synthesis with Metasketches. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. Association for Computing Machinery, New York, NY, USA, 775–788. https://doi.org/10.1145/2837614.2837666

Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and Direct Manipulation, Together at Last. *ACM SIGPLAN Notices* 51, 6 (June 2016), 341–354. https://doi.org/10.1145/2980983.2908103

Thierry Coquand. 1994. Infinite Objects in Type Theory. In *Types for Proofs and Programs (Lecture Notes in Computer Science)*, Henk Barendregt and Tobias Nipkow (Eds.). Springer, Berlin, Heidelberg, 62–78. https://doi.org/10.1007/3-540-58085-9_72

Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. 1980. Variations on the Common Subexpression Problem. *J. ACM* 27, 4 (Oct. 1980), 758–771. https://doi.org/10.1145/322217.322228

Kevin Ellis and Sumit Gulwani. 2017. Learning to Learn Programs from Examples: Going Beyond Program Structure. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*. International Joint Conferences on Artificial Intelligence Organization, Melbourne, Australia, 1638–1645. https://doi.org/10.24963/ijcai.2017/227

Azadeh Farzan, Danya Lette, and Victor Nicolet. 2022. Recursion Synthesis with Unrealizability Witnesses. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 244–259. https://doi.org/10.1145/3519939.3523726

Azadeh Farzan and Victor Nicolet. 2021. Counterexample-Guided Partial Bounding for Recursive Function Synthesis. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 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 (PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 229–239. https://doi.org/10.1145/2737924.2737977

Robert W. Floyd. 1967. Nondeterministic Algorithms. *J. ACM* 14, 4 (Oct. 1967), 636–644. https://doi.org/10.1145/321420.321422

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

Martin Hofmann. 2010. IGOR2 - an Analytical Inductive Functional Programming System: Tool Demo. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '10)*. Association for Computing Machinery, New York, NY, USA, 29–32. https://doi.org/10.1145/1706356.1706364

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 (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 944–959. https://doi.org/10.1145/3453483.3454087

Ruyi Ji, Jingtao Xia, Yingfei Xiong, and Zhenjiang Hu. 2021. Generalizable Synthesis through Unification. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (Oct. 2021), 1–28. https://doi.org/10.1145/3485544

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*. ACM, Indianapolis Indiana USA, 407–426. https://doi.org/10.1145/2509136.2509555

James Koppel, Zheng Guo, Edsko de Vries, Armando Solar-Lezama, and Nadia Polikarpova. 2022. Searching Entangled Program Spaces. *Proceedings of the ACM on Programming Languages* 6, ICFP (Aug. 2022), 91:23–91:51. https://doi.org/10.1145/3547622

Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. 2010. Complete Functional Synthesis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 316–329. https://doi.org/10.1145/1806596.1806632

Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. 2003. Programming by Demonstration Using Version Space Algebra. *Machine Learning* 53, 1 (Oct. 2003), 111–156. https://doi.org/10.1023/A:1025671410623

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 (PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 542–553. https://doi.org/10.1145/2594291.2594333

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 (Jan. 2021), 1–28. https://doi.org/10.1145/3434335

Woosuk Lee and Hangyeol Cho. 2023. Inductive Synthesis of Structurally Recursive Functional Programs from Non-recursive Expressions. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 70:2048–70:2078. https://doi.org/10.1145/3571263

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 (Aug. 2020), 109:1–109:29. https://doi.org/10.1145/3408991

Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. 2022. Bottom-up Synthesis of Recursive Functional Programs Using Angelic Execution. *Proceedings of the ACM on Programming Languages* 6, POPL (Jan. 2022), 21:1–21:29. https://doi.org/10.1145/3498682

Tom M. Mitchell. 1982. Generalization as Search. *Artificial Intelligence* 18, 2 (March 1982), 203–226. https://doi.org/10.1016/0004-3702(82)90040-6

Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 1–32. https://doi.org/10.1145/3290327

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

Saswat Padhi, Todd Millstein, Aditya Nori, and Rahul Sharma. 2019. Overfitting in Synthesis: Theory and Practice. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 315–334. https://doi.org/10.1007/978-3-030-25540-4_17

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. 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*. ACM, Pittsburgh PA 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 *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 404–415. https://doi.org/10.1109/ICSE.2017.44

Kensen Shi, Jacob Steinhardt, and Percy Liang. 2019. FrAngel: Component-Based Synthesis with Control Structures. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 1–29. https://doi.org/10.1145/3290386

Calvin Smith and Aws Albarghouthi. 2016. MapReduce Program Synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 326–340. https://doi.org/10.1145/2908080.2908102

Phillip D. Summers. 1977. A Methodology for LISP Program Construction from Examples. *J. ACM* 24, 1 (Jan. 1977), 161–175. https://doi.org/10.1145/321992.322002

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Synthesis of Data Completion Scripts Using Finite Tree Automata. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (Oct. 2017), 1–26. https://doi.org/10.1145/3133886

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2018a. Program Synthesis Using Abstraction Refinement. *Proceedings of the ACM on Programming Languages* 2, POPL (Jan. 2018), 1–30. https://doi.org/10.1145/3158151

Xinyu Wang, Sumit Gulwani, and Rishabh Singh. 2016. FIDEX: Filtering Spreadsheet Data Using Examples. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 195–213. https://doi.org/10.1145/2983990.2984030

Yuepeng Wang, Xinyu Wang, and Isil Dillig. 2018b. Relational Program Synthesis. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (Oct. 2018), 1–27. https://doi.org/10.1145/3276525

Yongwei Yuan, Arjun Radhakrishna, and Roopsha Samanta. 2023. *Artifact for "Trace-Guided Inductive Synthesis of Recursive Functional Programs"*. https://doi.org/10.5281/zenodo.7812616

Received 2022-11-10; accepted 2023-03-31