Synthesizing DSLs for Few-Shot Learning

PAUL KROGMEIER, University of Illinois, Urbana-Champaign, USA
P. MADHUSUDAN, University of Illinois, Urbana-Champaign, USA

We study the problem of synthesizing domain-specific languages (DSLs) for few-shot learning in symbolic domains. Given a base language and instances of few-shot learning problems, where each instance is split into training and testing datasets, the DSL synthesis problem asks for a grammar over the base language which guarantees that small expressions solving training datasets also solve corresponding testing datasets, where the notion of expression size varies as a parameter. Furthermore, the grammar must meet syntax constraints given as input. We prove that the problem is decidable for a class of languages whose semantics can be evaluated by tree automata and when expression size corresponds to parse tree depth in the grammar, and, furthermore, the grammars solving the problem correspond to a regular set of trees. We also prove decidability for a relaxation of DSL synthesis which asks for a grammar that meets syntax constraints and contains solutions to each learning instance. The proofs yield decision procedures based on tree automaton emptiness algorithms.

1 INTRODUCTION

In this work, we are interested in *few-shot learning of symbolic expressions*— learning symbolic classifiers in a logic that separate a given finite set of positive and negative examples or learning symbolic programs that compute a function matching a given finite set of input-output examples.

For a large class of concepts C, it is typically impossible to identify a target concept $c \in C$ from just a small set of samples S and hence succeed at few-shot learning. In practice, few-shot learning, such as program synthesis from examples, is successful because *researchers* identify a much smaller class of concepts $\mathcal{H}$, called the *hypothesis class*, and learn concepts from $\mathcal{H}$. The hypothesis class in symbolic learning is defined using a *language* (often referred to as a domain-specific language or DSL) that captures the more common and typical concepts in the domain. For few-shot learning, there is often also an *ordering* of concepts in $\mathcal{H}$, and few-shot learning algorithms find and report, appealing to Occam's razor, the smallest concept in $\mathcal{H}$ according to this ordering that is consistent with the samples S.

The literature on program synthesis/learning from input-output examples is replete with clever human design of DSLs. These DSLs are specially crafted by researchers for each application domain, accompanied by either an efficient learning algorithm that works for that hypothesis class or using generic program synthesis tools, e.g. [Gulwani 2011; Polozov and Gulwani 2015]. For example, to automatically complete the columns of a spreadsheet to match some given example strings, DSLs identify the most common string-manipulation functions that occur in spreadsheet programming [Gulwani 2011]. The SyGuS format (syntax-guided synthesis) for program synthesis makes this discipline of defining hypothesis classes explicit using *grammars* to define a DSL in which programs/expressions are synthesized. Recent work in semantics-guided synthesis supports specifying syntax for DSLs as well as semantics as part of the synthesis problem [Kim et al. 2021].

Formulation of DSL Synthesis. In this paper, we are interested in automatically synthesizing DSLs for few-shot learning, replacing work currently done by researchers. The first contribution of this paper is a *definition* of the problem of DSL synthesis for solving few-shot learning problems. We ask:

Authors' addresses: Paul Krogmeier, paulmk2@illinois.edu, Department of Computer Science, University of Illinois, Urbana-Champaign, USA; P. Madhusudan, madhu@illinois.edu, Department of Computer Science, University of Illinois, Urbana-Champaign, USA.

2024. ACM XXXX-XXXX/2024/4-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

What formulation of DSL synthesis facilitates few-shot learning in a domain?

Given an application domain D, we would like to formalize DSL synthesis for D itself using *learning*. Intuitively, we propose to learn DSLs from *instances of few-shot learning problems.*

Let us fix a base language using a grammar G over a finite signature that provides function, relation, and constant symbols, and where expressions in G have a fixed semantics.

Consider a finite training set of few-shot learning instances $\mathcal{I}$ obtained from a domain. We would like to learn a DSL $\mathcal{H}$, which includes a *syntax* for expressions and a *semantics* for expressions formalized using the base grammar G, that effectively solves each of the problems in $\mathcal{I}$. Each $p \in \mathcal{I}$ is itself a learning problem: it includes a set of training examples X_p and a set of testing examples Y_p. We require the synthesized DSL $\mathcal{H}$ to solve each of these problems $p \in I$ in the following sense: the *smallest* expressions in $\mathcal{H}$ (according to a fixed ordering on expressions) that are consistent with the training examples X_p must also be consistent with the testing examples Y_p.

We ask for the automated synthesis of DSLs using a small number of few-shot learning problems, replacing the human who has been typically responsible for this design. The bias in the DSL is informed by the typical few-shot learning problems in the domain that are presented. In addition to the few-shot learning instances, the input is also allowed to include a constraint $\mathcal{G}$, which serves as a meta-grammar that can constrain the DSL and provide additional syntactic bias on the semantics and syntax it uses. The DSL $\mathcal{H}$ must satisfy three properties in order to solve the problem:

- First, for each instance, $\mathcal{H}$ must be expressive enough to capture a concept c that solves the instance, in the sense that c is consistent with both the training and testing sets.
- Second, for each instance $p \in I$, consider the smallest concepts c that are expressible in $\mathcal{H}$, and minimal according to some ordering $<$, which satisfy all the training samples in X_p. Then c must also satisfy the testing samples in Y_p.
- Third, the definition of the hypothesis class $\mathcal{H}$ in terms of the base language must meet the syntactic constraints given by $\mathcal{G}$.

Intuitively, these conditions demand that for any concept expressible in the base language G that solves the training set X_p but does not solve the testing set Y_p (for some $p \in P$), the DSL $\mathcal{H}$ must either *disallow* expressing this concept or make sure that there are smaller concepts expressible in $\mathcal{H}$ that solve the training set X_p and testing set Y_p.

The first two conditions above need to be met for each few-shot learning instance in $\mathcal{I}$ for $\mathcal{H}$ to be a valid solution. The automated design of $\mathcal{H}$ hence needs to bias concepts that it expresses so that it captures the more "natural" concepts for the domain using simpler expressions.

Decidability of DSL Synthesis. The DSL synthesis problem for few-shot learning is a meta-synthesis problem (synthesizing a DSL that in turn solves a set of few-shot synthesis problems) and is algorithmically complex. A natural question that arises is whether there are natural and powerful subclasses where the problem is decidable. More precisely, for a given signature for defining the hypothesis class $\mathcal{H}$, we would like algorithms that, given a set of few-shot learning instances $\mathcal{I}$ and syntax constraint $\mathcal{G}$, either synthesize an $\mathcal{H}$ that solves the instances and satisfies $\mathcal{G}$ or report that no solution exists. We allow the semantics of the DSLs, defined in terms of an existing language with grammar G, to be of *arbitrary length*, which makes decidability nontrivial.

We prove that the DSL synthesis problem is indeed decidable for a class of languages where the semantics of expressions can be *evaluated* bottom-up using finite memory. The technique that we use to establish decidability relies on *tree automata*— we show that the class of trees encoding DSLs which solve the few-shot learning instances is in fact a regular set of trees. Our result builds upon recent techniques to learn expressions in languages that can be evaluated bottom-up using memory which can depend arbitrarily on the sizes of examples, but which is independent of expression size [Krogmeier and Madhusudan 2022, 2023]. Our constructions are much more complex (both

conceptually and in terms of time complexity) than these previous constructions, as the synthesis of suitable DSLs involves finding *minimal* expressions that witness the solvability of each given few-shot learning instance. In particular, we need to use alternating quantification on trees to capture the fact that there *exists* a solution to each instance such that *all* other smaller expressions do not solve the training set. As far as we know, this is the first decidability result on DSL synthesis.

Decidability of Grammar Synthesis. We also study a relaxation of the DSL synthesis problem, which we call grammar synthesis. Given a set of few-shot learning instances $\mathcal{I}$, the grammar synthesis problem asks for a DSL $\mathcal{H}$ which contains a concept consistent with each $p \in \mathcal{I}$, with no requirement on the ordering of consistent expressions. We prove grammar synthesis is decidable for the same class of base languages as for DSL synthesis. The proof is similar in that it involves constructing tree automata which capture the set of trees encoding DSLs which solve each input learning instance, and thus the set of solutions corresponds to a regular set of trees. It is also a considerably simpler construction, as the specification is simpler— it no longer involves alternating quantification on trees, and the decision procedure we obtain has lower complexity as a result.

Contributions. In summary, this paper makes the following contributions:

- A novel definition of DSL synthesis which asks for a hypothesis class that biases toward few-shot learning in a domain, using few-shot learning instances as input.
- A decidability result for DSL synthesis over a powerful subclass of base languages.
- A natural relaxation of the DSL synthesis problem, called grammar synthesis, and a corresponding decidability result over the same subclass.

The paper is organized as follows. In Section 2, we explore the DSL synthesis problem with some simple illustrative examples. In Section 3, we review background and introduce some concepts related to the problem formulation. In Section 4, we present our formulation of DSL synthesis. In Section 5, we prove decidability of DSL synthesis for a class of base languages whose semantics can be computed by tree automata and when expression order is given by parse tree depth. In Section 6, we introduce a natural relaxation of DSL synthesis, which we call grammar synthesis, and prove its decidability. Section 7 reviews related work and Section 8 concludes.

2 EXAMPLES

In this section we introduce the *DSL synthesis problem* with some examples.

2.1 Linear Classifiers

Suppose we have some datasets of points in the plane

$$D^1 = \{(x^1_1, y^1_1), (x^1_2, y^1_2), \dots, (x^1_{n_1}, y^1_n)\} \subseteq \mathbb{R}^2, \quad \dots, \quad D^k = \{(x^k_1, y^k_1), (x^k_2, y^k_2), \dots, (x^k_{n_k}, y^k_{n_k})\} \subseteq \mathbb{R}^2$$

and we want to fit a function to each set of points. Suppose for each dataset D^i, for some numbers $a_i, b_i \in \mathbb{R}$, the dataset D^i obeys

$$y_j = a_i x_j + b_i, \quad \text{for all } 1 \leq j \leq n_i.$$

If we do not know that a linear expression will do, we might search in a space of numerical expressions built out of both linear and nonlinear operations. For example, suppose we choose to search in the space of expressions defined by the following grammar.

$$\begin{aligned} S &::= \; Nx + S \;\mid\; N \;\mid\; Nx^2 \;\mid\; Nx^3 \;\mid\; \cdots \;\mid\; Nx^c \\ N &::= \; n \in \mathbb{R} \end{aligned}$$

Of course, we can find expressions in this grammar to fit each dataset D^i, and for large values of c we can fit many datasets beyond that. But, since two points in the plane determine a line, we might imagine a better search space for this specific domain would collapse to a single viable concept after we see just two examples.

In this paper, we introduce a *DSL synthesis problem* to explore how hypothesis classes can be synthesized to solve few-shot learning problems. In this example, DSL synthesis asks for a hypothesis class of arithmetic expressions which can be used for few-shot learning by enumerating small expressions.

Specifically, we would like a DSL such that, given a set of few-shot learning instances $\{D^i\}$, if we *enumerate expressions in order of increasing complexity*, the first consistent expression we find for each D^i should generalize well. Let us imagine splitting the datasets D^i into two sets of examples, a training set X^i and a testing set Y^i. Let us say an expression that is consistent with X^i *generalizes* on the problem D^i if it is also consistent with Y^i.

The DSL defined by the grammar above is not well tuned in this sense for linear datasets. For example, suppose the first two datasets are $D^1 = \{(0,0),(2,4),(4,8)\}$ and $D^2 = \{(0,0),(2,8),(3,12)\}$, split into

$$X^1 = \{(0,0),(2,4)\}, \quad Y^1 = \{(4,8)\}$$
$$\text{and} \;\; X^2 = \{(0,0),(2,8)\}, \quad Y^2 = \{(3,12)\}.$$

The least complex expression consistent with X^1, if we measure complexity by number of nonterminals in a smallest parse tree, is in fact the expression x^2, which has a parse tree of size 2 in the grammar. This expression *does not generalize* to Y^1 because $4^2 = 16 \neq 8$. And the simplest expression consistent with X^2 is the expression x^3, also of size 2, which fails to generalize on Y^2 because $3^3 = 27 \neq 12$. Perhaps the other training datasets X^i can be fit by other small nonlinear functions. To address this, we could synthesize a grammar for arithmetic expressions which perhaps does not allow the nonlinear operations at all, or perhaps it allows them, but enables linear expressions to be expressed more succinctly. For example, the DSL specified by the grammar below solves the problem by making the nonlinear operations more complex in the grammar.

$$S ::= \; Nx + N \;\mid\; S \times S \qquad N ::= \; n \in \mathbb{R}$$

In this DSL, the least complex solutions for the training sets X^1 and X^2 are $2x$ and $4x$, with size 3, each of which is also consistent with its test set.

2.2 LTL from FOL

Suppose our base language is first-order logic (FOL), and we are trying to solve classification problems over labeled finite words over an alphabet Σ. Our base grammar, shown below, might allow us a handful of variables V, along with an ordering on positions in words, equality on positions, and monadic predicates $P(x)$, one for each $p \in \Sigma$.

$$\varphi ::= P(x) \;\mid\; x < x' \;\mid\; \varphi \wedge \varphi' \;\mid\; \varphi \vee \varphi' \;\mid\; \neg\varphi \;\mid\; \exists x.\, \varphi \;\mid\; \forall x.\, \varphi$$

Consider few-shot learning problems where the relevant properties are naturally expressed in linear temporal logic (LTL) over an alphabet of propositions $\Sigma = \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$. For example, suppose we have the following few-shot learning instance split into a training set X and a testing set Y:

$$X = \{(+, \mathrm{ab}), (+, \mathrm{aab})\}$$
$$Y = \{(+, \mathrm{aaab}), (-, \mathrm{aaa}), (-, \mathrm{ccb}), (-, \mathrm{cab})\}$$

In this setting, we might say a sentence φ from the base grammar is *consistent* with an example $(label, w)$ if

$$\begin{aligned} w &\models \varphi \quad \text{if } label = + \\ w &\not\models \varphi \quad \text{if } label = - \end{aligned}$$

Within this base language, there are many simple concepts consistent with the training set X, e.g.,

$$\exists x.\, A(x),\ \ \exists x.\, B(x),\ \ \text{or}\ \ \exists x.\, \exists y.\, A(x) \land B(y),$$

which are not consistent with the testing set Y.

A better hypothesis class for few-shot learning in this setting might be one which defines the until operator $\psi'\, U\, \psi$ from LTL:

$$\begin{aligned} \varphi &::= \exists y.\, \psi \land \big(\forall x.\, \neg(x < y) \lor \psi'\big) \\ \psi, \psi' &::= P(x) \ \mid\ x < y \ \mid\ \neg\psi \ \mid\ \psi \land \psi' \ \mid\ \psi \lor \psi', \end{aligned}$$

where ψ, ψ' above range over Boolean combinations of atomic formulas over all propositions and variables. With respect to a DSL defined by the grammar above, the sentence

$$\exists y.\, B(y) \land \big(\forall x.\, \neg(x < y) \lor A(x)\big),$$

which expresses "a is true until b", is less complex than in the base language (has a smaller parse tree), and it is also consistent with Y.

2.3 Stutter-Invariant Temporal Properties

Stutter invariance [Lamport 1983] is a concept from temporal logic that describes properties of a system which depend only on its observable state variables and not on the number of time steps that it stays in a given state. For example, the property that *all requests are eventually responded to*, expressed in LTL as

$$\varphi \coloneqq \mathsf{G}\,(\mathsf{Req} \rightarrow \mathsf{F}\,(\mathsf{Resp})),$$

is stutter invariant because any trajectory that satisfies φ will still satisfy φ if we *stutter* it any number of times, where stutter means to repeat some of the existing letters, e.g., *abbc* is obtained from *abc* by stuttering the second position one time. Any trajectory in $(u \cdot \mathsf{Req} \cdot v \cdot \mathsf{Resp})^{\omega}$, with $u, v \in \Sigma^*$, satisfies φ, and so too do any of the stuttered versions. Similarly, if a trajectory does not satisfy φ then stuttering does not change that.

If stutter invariance is important in a domain, we might use testing datasets obtained by stuttering the examples in training datasets to bias toward a hypothesis class that cannot detect stuttering. For example, from a base grammar for LTL formulas

$$\varphi ::= p \in P \ \mid\ \mathsf{X}\varphi \ \mid\ \mathsf{F}\varphi \ \mid\ \mathsf{G}\varphi \ \mid\ \varphi \mathsf{U} \varphi' \ \mid\ \varphi \land \varphi' \ \mid\ \varphi \lor \varphi' \ \mid\ \neg\varphi$$

we might prefer a DSL that omits the next time operator

$$\varphi ::= p \in P \ \mid\ \mathsf{F}\varphi \ \mid\ \mathsf{G}\varphi \ \mid\ \varphi \mathsf{U} \varphi' \ \mid\ \varphi \land \varphi' \ \mid\ \varphi \lor \varphi' \ \mid\ \neg\varphi$$

as this language is equally expressive as the former for stutter-invariant properties [Peled and Wilke 1997].

3 PRELIMINARIES

In this section, we review standard concepts like ranked alphabets, tree grammars, and tree automata. Next we describe meta-grammars, including our encoding of grammars as trees, base languages, consistency, examples, and expression size and depth with respect to a grammar.

3.1 Alphabets and Grammars

Definition 3.1 (Ranked alphabet). A ranked alphabet Δ is a set of symbols with arities given by a function arity : $\Delta \to \mathbb{N}$. We write Δ^i for the subset of Δ that has arity i. We use T_Δ to denote the smallest set of terms containing the nullary symbols in Δ and closed under forming new terms using symbols of larger arity. For a set of nullary symbols X disjoint from Δ^0 we write $T_\Delta(X)$ to mean $T_{\Delta \cup X}$. We omit Δ from T_Δ and $T_\Delta(X)$ when it is clear in context.

Definition 3.2 (Tree grammar). A tree grammar is a tuple $G = (S, N, \Delta, P)$, where N is a finite set of nonterminal symbols, $S \in N$ is the starting nonterminal symbol, Δ is a ranked alphabet, and $P \subseteq N \times T(N)$ is a finite set of rules. Rules (N, t) can be written also as $N \leftarrow t$.

Example 3.3. We use vertical bars in the standard way to depict a tree grammar by listing its productions. Here is one for positive propositional logic formulas in conjuctive normal form over variables x_i.

$$S \leftarrow \wedge(D, S) \mid D \mid \mathsf{true}$$
$$D \leftarrow \vee(X, D) \mid X \mid \mathsf{false}$$
$$X \leftarrow x_1 \mid \cdots \mid x_k$$

The *language* $L(G) \subseteq T_\Delta$ of a tree grammar G is defined in the standard way as a least fixpoint. We often write $t \in G$ instead of $t \in L(G)$ to refer to a term in the language of G. In the remainder of the paper, when we say *grammar* we mean a *tree* grammar. We also use the words *tree* and *term* interchangeably.

3.2 Tree Automata

We make use of *tree automata* in Sections 5 and 6. For background on tree automata we refer the reader to [Comon et al. 2007], but we highlight some salient aspects here.

We make use of *two-way alternating tree automata.* Intuitively, such automata process an input tree by traversing it both *up* and *down* while branching universally in addition to existentially. The transitions are given by Boolean formulae which describe the valid actions the automaton can take to build an accepting run. For a symbol h and state set Q, with $q \in Q$, the transition $\delta(q, h)$ is a positive Boolean formula φ over atoms $Q \times \{-1, 0, \dots, \mathsf{arity}(h)\}$. The automaton makes a transition by picking a satisfying assignment for the atoms and continuing from the nearby nodes and states designated by true atoms. For example, the formula

$$(q_1, -1) \wedge (q_2, -1) \vee (q_2, 0) \wedge (q_3, 1)$$

requires that the automaton *either* continues at the parent node (designated by -1) from both q_1 and q_2 *or* continues at the current node (designated by 0) in q_2 and the first child in q_3. We adopt more intuitive notation at times for the directions in the tree, e.g., we use **up**, **stay**, **left**, and **right** for -1, 0, 1, and 2.

We will assume for convenience that both non-deterministic top-down tree automata and two-way alternating tree automata have the form $A = (Q, \Delta, I, \delta)$, with states Q, alphabet Δ, initial states $I \subseteq Q$, and δ a transition formula as described above. In the case of non-deterministic automata, the transition formulas are restricted to disallow any stationary moves, moves to the parent, or universal branching, i.e., multiple states required at a single node. Under these restrictions, there is a close connection between tree automata and tree grammars, as a grammar rule $N \leftarrow f(N_1, \dots, N_k) \in P$ corresponds to a transition $\delta(N, f) = (N_1, 1) \wedge \cdots \wedge (N_k, k)$, and given any tree grammar G we can compute in polynomial time a non-deterministic top-down tree automaton $A(G)$ with $L(A(G)) = L(G)$.

All automata we use in this work have acceptance defined in terms of the existence of a run on an input tree. We refer the reader to [Comon et al. 2007, Section 7] for background on this presentation of tree automata and for the notion of a run.

3.3 Encoding Grammars as Trees

We will be defining tree automata whose inputs are trees that encode tree grammars. There are many natural ways to encode a tree grammar itself as a tree; here we describe one such way which we assume in our proofs.

To encode a grammar $G = (S, N, \Delta, P)$ as a tree t we arrange its productions (N_i, α) along the topmost right-going spine of t, with each α hanging to the left, as shown in Figure 1 for $\Delta = \{a^0, h^1, g^2\}$, $N = \{N_1, N_2\}$, and $S = N_1$. We use the symbol "root" to indicate the root of the tree and "end" to indicate there are no more productions at the right edge of the tree. We use symbols "lhs_{N_i}" and "rhs_{N_i}" to distinguish between occurrences of nonterminals in the left-hand and right-hand sides of a production. We write $\Gamma(\Delta, N)$ to denote *grammar alphabets* which encode tree grammars over alphabet Δ and nonterminals N.

Definition 3.4 (Grammar alphabet). Given a ranked alphabet Δ and a set of nonterminal symbols N, we write $\Gamma(\Delta, N)$ for the alphabet $\Delta \sqcup \{\mathsf{root}^1, \mathsf{end}^0\} \sqcup \{\mathsf{lhs}^2_{N_i}, \mathsf{rhs}^0_{N_i} \,:\, N_i \in N\}$.

Formally, we define a mapping from grammars to tree encodings, which we call *grammar trees*, and a mapping from grammar trees to grammars.

The grammar tree for a grammar $G = (S, N, \Delta, P)$, where we assume the productions are ordered in a list as $P = \langle P_1, P_2, \ldots, P_s \rangle$, is given by $\mathsf{enc}(G) = \mathsf{root}(\mathsf{enc}_p(P))$, where the spine of productions is computed recursively on the list P as follows.

$$
\begin{array}{lcl}
\mathsf{enc}_p(\langle (N_i, \alpha), L \rangle) & = & \mathsf{lhs}_{N_i}(\mathsf{enc}_t(\alpha), \mathsf{enc}_p(L)) \\
\mathsf{enc}_p(\langle\rangle) & = & \mathsf{end} \\
\mathsf{enc}_t(f(t_1, \ldots, t_r)) & = & f(\mathsf{enc}_t(t_1), \ldots, \mathsf{enc}_t(t_r)) \\
 & & \text{where } f \in \Delta, \mathsf{arity}(f) = r \\
\mathsf{enc}_t(N_i) & = & \mathsf{rhs}_{N_i}
\end{array}
$$

The grammar G corresponding to a grammar tree t of the form $\mathsf{root}(\mathsf{lhs}_{N_i}(x, y))$ over alphabet $\Gamma(\Delta, N)$, is given by

$$
\mathsf{dec}(t) = (N_i, N, \Delta, \langle (N_i, \mathsf{dec}_t(x)), \mathsf{dec}_p(y) \rangle),
$$

where dec_t and dec_p are computed recursively as follows.

$$
\begin{array}{lcl}
\mathsf{dec}_p(\mathsf{lhs}_{N_i}(x, y)) & = & \langle (N_i, \mathsf{dec}_t(x)), \mathsf{dec}_p(y) \rangle \\
\mathsf{dec}_p(\mathsf{end}) & = & \langle\rangle \\
\mathsf{dec}_t(f(x_1, \ldots, x_r)) & = & f(\mathsf{dec}_t(x_1), \ldots, \mathsf{dec}_t(x_r)) \\
 & & \text{where } f \in \Delta, \mathsf{arity}(f) = r \\
\mathsf{dec}_t(\mathsf{rhs}_{N_i}) & = & N_i
\end{array}
$$

We sometimes elide the distinction between a *grammar* and its encoding as a *grammar tree.*

3.4 Meta-Grammars and Extensions

The DSL synthesis problem is parameterized by a base language with grammar G' and the goal is to synthesize a new grammar G that can use nonterminals of G' while also defining productions for new nonterminals that do not appear in G'.

We will use regular constraints on grammar trees G to specify inductive bias in the DSL synthesis problem. The constraints are presented as meta-grammars $\mathcal{G}$, i.e., grammars that constrain grammar trees. For the synthesis of a grammar G that uses nonterminals N defined over a base language with grammar $G' = (S', N', \Delta, P')$, with $N' \subseteq N$, the meta-grammar is over the alphabet

Grammar $\mathcal{G}$ over $\Gamma(\Delta, N)$:

$$
\begin{array}{rcll}
S & \leftarrow & \mathsf{root}(\mathit{Prod}) & \\
\mathit{Prod} & \leftarrow & \mathsf{lhs}_{N_i}(\mathit{Term}, \mathit{Prod}) & N_i \in N \\
 & \mid & \mathsf{end} & \\
\mathit{Term} & \leftarrow & g(\mathit{Term}, \mathit{Term}) & \\
 & \mid & h(\mathit{Term}) & \\
 & \mid & a & \\
 & \mid & \mathsf{rhs}_{N_i} & N_i \in N
\end{array}
$$

Grammar G over Δ:

$$
\begin{array}{l}
N_1 \leftarrow h(N_2) \\
N_2 \leftarrow h(N_1) \mid g(a, a)
\end{array}
$$

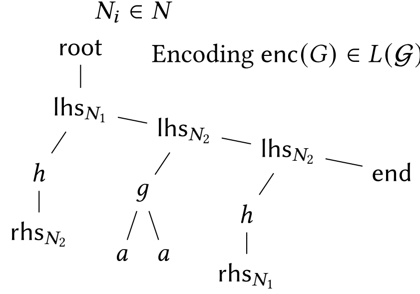

Fig. 1. Encoding tree grammars as trees. (Top right) A tree grammar G over $\Delta = \{a^0, h^1, g^2\}$ and nonterminals $N = \{N_1, N_2\}$ and (bottom right) its encoding as a tree $\mathsf{enc}(G)$ over the alphabet $\Gamma(\Delta, N)$. (Left) A tree grammar $\mathcal{G}$ over alphabet $\Gamma(\Delta, N)$ with $\mathsf{enc}(G) \in L(\mathcal{G})$.

$\Gamma(\Delta, N)$. The problem then asks for the synthesis of a grammar $G \in L(\mathcal{G})$ such that the extension of G with G' is a solution to the problem, with extension defined as follows.

Definition 3.5 (Grammar extension). Given a grammar $G = (S, N, \Delta, P)$ and a grammar $G' = (S', N', \Delta, P')$ we define the *extension* of G with G' by $\mathsf{ext}(G, G') \coloneqq (S, N \cup N', \Delta, P \cup P')$.

3.5 Base Language, Examples, and Consistency

3.5.1 Base Language. We consider synthesizing DSLs defined over a *base language*. To specify the base language we specify a grammar $G' = (S', N', \Delta, P')$, a set $\mathcal{M}$ from which *examples* are drawn, and a predicate called consistent $\subseteq L(G') \times \mathcal{M}$, which captures whether an expression e is consistent with an example $M \in \mathcal{M}$, written $\mathsf{consistent}(e, M)$. This predicate abstracts away details of specific base languages while keeping the information relevant for proofs of our main results in Sections 5 and 6.

Example 3.6 (Propositional Logic). If our base language consists of a grammar for propositional logic formulas φ over variables X, our examples could be variable assignments $\alpha : X \longrightarrow \{0, 1\}$ labeled positive or negative. If we denote positive and negative examples by $p(\alpha)$ and $n(\alpha)$, respectively, then consistency might be defined as

$$
\begin{array}{l}
\mathsf{consistent}(\varphi, p(\alpha)) \Leftrightarrow \alpha \models \varphi \quad \text{and} \\
\mathsf{consistent}(\varphi, n(\alpha)) \Leftrightarrow \alpha \not\models \varphi.
\end{array}
$$

Example 3.7 (Regular Expressions). For a base language of regular expressions r over $\Sigma = \{a, b, c\}$, our examples can be labeled words $p(w), n(w)$ with $w \in \Sigma^*$. Here, consistency can be defined as

$$
\begin{array}{l}
\mathsf{consistent}(r, p(w)) \Leftrightarrow w \in L(r) \quad \text{and} \\
\mathsf{consistent}(r, n(w)) \Leftrightarrow w \notin L(r).
\end{array}
$$

Example 3.8 (Linear Integer Arithmetic). For a base language of linear arithmetic expressions in two variables x, y interpreted over the integers, the examples might be input-output pairs, i.e., $((a, b), o)$, for $a, b, o \in \mathbb{Z}$. Here, consistency can be defined as

$$\mathsf{consistent}(e, ((a, b), o)) \Leftrightarrow \mathsf{LIA}, \{x \mapsto a, y \mapsto b\} \models e = o,$$

where LIA stands for the theory of linear integer arithmetic.

3.5.2 Language Semantics Computable by Tree Automata. For a large class of base languages, called finite-aspect checkable (FAC) languages in [Krogmeier and Madhusudan 2023], for each example $M \in \mathcal{M}$, the predicate consistent, when specialized to M, can be computed by tree automata whose size is only a function of $|M|$. For such a language, for every $M \in \mathcal{M}$ there exist tree automata A_M and $A_{\neg M}$ with $L(A_M) = \{t \in L(G') : \mathsf{consistent}(t, M)\}$ and $L(A_{\neg M}) = \{t \in L(G') : \neg\mathsf{consistent}(t, M)\}$. We use these *example automata* in our proofs in Sections 5 and 6.

3.6 Expression Size and Depth

Our formulation of the DSL synthesis problem depends on a grammar-dependent ordering over expressions which captures the preference given to individual expressions by a grammar. We describe here two natural orderings based on size and depth.

Definition 3.9 (Expression Size in a Grammar). Given a grammar G and an expression $e \in G$, we define the *size* of e to be the minimum length of any derivation of e using the rules of G. We denote this by $\mathsf{size}(e, G)$.

Example 3.10. The expression $\mathsf{f}^{100}(a)$ has size 101 with respect to this grammar:

$$S \leftarrow \mathsf{f}(S) \mid a$$

but it has size 3 with respect to this grammar:

$$S \leftarrow \mathsf{f}^{50}(S) \mid a,$$

since we can derive $\mathsf{f}^{100}(a)$ with $S \Rightarrow \mathsf{f}^{50}(S) \Rightarrow \mathsf{f}^{50}(\mathsf{f}^{50}(S)) \Rightarrow \mathsf{f}^{50}(\mathsf{f}^{50}(a)) = \mathsf{f}^{100}(a)$.

Definition 3.11 (Expression Depth in a Grammar). Given a grammar G and an expression $e \in G$, we define the *depth* of e to be the minimum over all parse trees T for e of the maximum number of nonterminals encountered along any root-to-leaf path in T. We denote this by $\mathsf{depth}(e, G)$.

Example 3.12. The expression $x + x + x + x$ has depth 3 with respect to this grammar

$$S \leftarrow S + S \mid x,$$

and it has depth 4 with respect to this grammar

$$S \leftarrow x + S \mid x.$$

Note also that x has depth 1.

4 THE DSL SYNTHESIS PROBLEM

In this work we study the problem of synthesizing DSLs given *instances of few-shot learning problems* as inputs.

Definition 4.1 (Learning Instance). A *learning instance* is a pair (X, Y) consisting of a set X of *training* examples and a set Y of *testing* examples.

The DSL synthesis problem is parameterized by a base language in terms of which the DSL can be defined. As described in Section 3.5.1, the base language consists of an expression grammar G', a domain of examples $\mathcal{M}$, and a domain-specific predicate $\mathsf{consistent}(e, M)$ which captures whether an expression e satisfies or does not satisfy an example M.

We say an expression $e \in G'$ *solves* a set of examples $X \subseteq \mathcal{M}$ if it is consistent with each example in X, and we use the notation

$$\mathsf{solves}(e, X) \coloneqq \bigwedge_{M \in X} \mathsf{consistent}(e, M).$$

Similarly, we say an expression e *solves* or *generalizes on* an instance $I = (X, Y)$ if it solves $X \cup Y$, written $\mathsf{solves}(e, I)$. If an expression e solves X but does not solve $I = (X, Y)$ we say that e *does not generalize* to Y.

Example 4.2. The regular expression abc fails to generalize for $I = (\{p(abc), n(abb)\}, \{p(abbc)\})$ because it matches all positive words and no negative words in the training set, but it does not match the positive word in the testing set. In contrast, we do not say the expression $ab(b + c)$ fails to generalize because it does not even solve the training set.

4.1 Problem Definition

Here we define the DSL synthesis problem.

Problem (DSL synthesis)**.**
Parameters:

- Finite set of nonterminals N
- Base language with $G' = (S', N', \Delta, P')$ and $N' \subseteq N$
- Predicate $\mathsf{order}(e, e', G)$[1]

Input:

- Learning instances $I_1, \ldots, I_l$
- Meta-grammar $\mathcal{G}$ over $\Gamma(\Delta, N)$

Output: A grammar $G = (S, N, \Delta, P)$ such that:

(1) for each instance I, there is an expression $e \in \mathsf{ext}(G, G')$ such that $\mathsf{solves}(e, I)$ holds, and for all $e' \in \mathsf{ext}(G, G')$ which fail to generalize on I, $\mathsf{order}(e, e', \mathsf{ext}(G, G'))$ holds
(2) it satisfies constraints imposed by $\mathcal{G}$, i.e., $\mathsf{enc}(G) \in L(\mathcal{G})$

We write $\mathsf{solves}(G, I)$ if Item 1 holds above for instance I and grammar G.

Solutions to DSL synthesis are grammars that make desirable generalizing expressions appear early in the order and non-generalizing expressions appear later in the order. In some natural cases the order predicate is defined using an underlying strict total order, like parse tree depth or size, as described in Section 3.6. In these cases, a grammar that solves DSL synthesis can be used to solve learning instances by enumerating expressions in increasing order and selecting the first one that solves the training set. Let us see an example for propositional logic.

Example 4.3 (Propositional Logic). Consider the following DSL synthesis problem over a base language of propositional logic with grammar G' given by

$$S \leftarrow \wedge(S, S) \;\mid\; \vee(S, S) \;\mid\; \neg(S) \;\mid\; x_1 \;\mid\; \cdots \;\mid\; x_k \;\mid\; \mathsf{true} \;\mid\; \mathsf{false},$$

[1]The order predicate can be interpreted to mean e is preferred over e' by grammar G.

and with nonterminals $N = \{S, N_1, N_2\}$ and $N' = \{S\}$, and with no meta-grammar constraint. It consists of a single instance

$$X = \{p(\{x_2\}), n(\{x_1, x_2\})\}, Y = \{n(\emptyset)\},$$

where variable assignments are represented as sets of variables assigned true. In this example, the predicate $\mathsf{order}(e, e', G)$ is defined as $\mathsf{size}(e, G) < \mathsf{size}(e', G)$. Observe that we could synthesize $G = G'$ by picking S as the starting nonterminal and not adding any new productions for it. This is *not a solution*. Indeed, the smallest expression from G' that solves X is $\neg x_1$, with $\mathsf{size}(\neg x_1, G') = 2$, and it does not solve Y. A solution for $X \cup Y$ like $\neg x_1 \wedge x_2$ is too large, with $\mathsf{size}(\neg x_1 \wedge x_2, G') = 4$. But we can prefer $\neg x_1 \wedge x_2$ over $\neg x_1$ by synthesizing the grammar $G = (N_1, \{N_1, N_2, S\}, \Delta, P)$, where P is the set of productions below.

$$\begin{aligned} N_1 &\leftarrow N_1 \wedge N_1 \mid N_1 \vee N_1 \mid \neg x_1 \wedge N_2, \\ N_2 &\leftarrow x_1 \mid \cdots \mid x_k \end{aligned}$$

In the grammar $\mathsf{ext}(G, G')$, the smallest $X \cup Y$ solution is $\neg x_1 \wedge x_2$, with $\mathsf{size}(\neg x_1 \wedge x_2, \mathsf{ext}(G, G')) = 2$. And in fact, $\mathsf{ext}(G, G')$ does not allow any non-generalizing expressions, so G is a solution.

Example 4.4 (Arithmetic). Consider the DSL synthesis problem with nonterminals $N = N' = \{S\}$ over a base language of arithmetic expressions on a single variable x with grammar G' below.

$$S \leftarrow S + S \mid S - S \mid S \times S \mid -S \mid x \mid 1 \mid 2 \mid 3 \mid 4$$

There is one instance consisting of two input-output examples $X = (1, -1)$, $Y = (4, 2)$. If $\mathsf{order}(e, e', G)$ is defined by $\mathsf{depth}(e, G) \leq \mathsf{depth}(e', G)$, then the grammar G' itself is a solution. Although the non-generalizing expression $-x$ has a smaller parse tree in terms of size, it has the same parse tree depth as the generalizing expression $x - 2$, namely $\mathsf{depth}(-x, \mathsf{ext}(G', G')) = \mathsf{depth}(x - 2, \mathsf{ext}(G', G')) = 2$, and there are no non-generalizing expressions that are shallower.

5 DECIDABLE DSL SYNTHESIS

In this section we show that DSL synthesis is *decidable* over languages whose semantics can be computed by finite tree automata and when the expression order is given by parse tree depth. We consider the case when $\mathsf{order}(e, e', G)$ is defined by $\mathsf{depth}(e, G) \leq \mathsf{depth}(e', G)$, but the proof easily adapts to the strict case, i.e., to $\mathsf{depth}(e, G) < \mathsf{depth}(e', G)$ as well. Furthermore, we show that, via an encoding of grammars as trees, the set of solutions to DSL synthesis in fact corresponds to a regular set of trees.

The proof involves construction of a tree automaton A that reads finite trees which represent grammars. We will design A so that it accepts precisely those grammars which are solutions to a given DSL synthesis problem. Existence and synthesis of solutions is then accomplished with standard algorithms for checking emptiness of $L(A)$. A detailed construction of A is given in Section 5.2, but we begin with some intuition in Section 5.1 about *equivalence of grammars*, which paves the way for an automaton to evaluate arbitrarily large grammars using state that is bounded by the size of input learning instances.

5.1 Grammar Equivalence, Recursion Tables

The existence of a finite tree automaton A that accepts precisely those (trees representing) grammars that solve a given DSL synthesis problem implies that, for any learning instances $I_1, \ldots, I_l$, grammars can be divided up into finitely-many equivalence classes based on how they behave over each I_i. Let us think about what would make two distinct grammars G_1 and G_2 equivalent with respect to

$I_1, \ldots, I_l$. For the purposes of DSL synthesis, whether G_1 and G_2 are equivalent on these inputs will depend on the specific examples contained in each instance I_i. To simplify things, let us consider how the grammars might behave over a single example structure M with expressions interpreted over its domain D^M.

Suppose $G_1 = (N_1, \{N_1, N_2\}, \Delta, P_1)$ and $G_2 = (N_1, \{N_1, N_2\}, \Delta, P_2)$, with $\Delta = \{h^1, a^0\}$. Now consider a table whose entries are subsets of D and whose columns and rows are indexed by nonterminals and increasing integers, respectively. Suppose that $D^M = \{1, 2, 3, 4\}$, and the symbols h and a are interpreted as $h^M(1) = 2, h^M(2) = 3, h^M(3) = 4, h^M(4) = 1$, and $a^M = 1$. Suppose G_1 has productions

$$N_1 \leftarrow h(a) \ \mid \ h(N_2), \quad N_2 \leftarrow h(N_1).$$

Consider the simultaneous least fixpoint that defines $L(G_1)$ as a set of Δ-terms. Though it is an infinite set, if we consider terms modulo equivalence in M, then there are finitely-many equivalence classes, and the fixpoint computation needs only four steps to reach a fixpoint. Beyond depth four, expressions of G_1 repeat themselves on M. Figure 2 shows the stages of the fixpoint computation in a table, where the entry at row i and column j contains the set of new domain elements achieved in stage i for nonterminal j. Now suppose G_2 has productions

$$N_1 \leftarrow h^5(a) \ \mid \ h^5(N_2), \quad N_2 \leftarrow h^5(N_1).$$

If we build the table for G_2 we find it is identical to the table for G_1. These tables give us a notion of equivalence that captures whether two grammars have the same expressive power, parameterized by *parse tree depth*, over fixed structures. This notion enables us to give a tree automata-based solution for the DSL synthesis problem when the order on expressions is given by parse tree depth. We will use such *recursion tables*, albeit with a more complex domain D, in the automaton construction to come.

M: $Domain(M) = \{1, 2, 3, 4\}$
$h^M(1) = 2, h^M(2) = 3,$
$h^M(3) = 4, h^M(4) = 1$
$a^M = 1$

G_1: $N_1 \leftarrow h(a) \mid h(N_2)$
$N_2 \leftarrow h(N_1)$

G_2: $N_1 \leftarrow h^5(a) \mid h^5(N_2)$
$N_2 \leftarrow h^5(N_1)$

stage	N_1	N_2
0	$\emptyset$	$\emptyset$
1	$\{2\}$	$\emptyset$
2	$\emptyset$	$\{3\}$
3	$\{4\}$	$\emptyset$
4	$\emptyset$	$\{1\}$
5	$\emptyset$	$\emptyset$

Fig. 2. A recursion table for the grammar G_1 interpreted over M. Row i column j contains the domain values of M that are first reached by a term at depth i in G_1. The set of terms $L(G_1)$ is infinite but, modulo equivalence on M, the language is completely explored after stage four. This same table is obtained for G_2 interpreted over M.

5.2 Automaton Construction

The main aspect of our construction is an automaton A_I that accepts grammars that, when combined with the base grammar, solve an instance $I = (X, Y)$. Our final automaton A will involve a product over instances I of the automata A_I. For each example $M \in X \cup Y$, we assume the existence of

non-deterministic top-down tree automata A_M and $A_{\neg M}$ over alphabet Δ whose languages are

$$L(A_M) = \{e \in L(G') : \mathsf{consistent}(e, M)\} \text{ and}$$
$$L(A_{\neg M}) = \{e \in L(G') : \neg\mathsf{consistent}(e, M)\},$$

i.e., the sets of expressions in the base language which are consistent, or inconsistent, with the example. Let us call these *example automata*. If we can in fact construct such automata for a given base language, then our proof will apply. Using example automata, we can now define *instance automata* A_1^I and A_2^I:

$$A_1^I := \bigtimes_{M \in X \cup Y} A_M \qquad\qquad A_2^I := \left(\bigtimes_{M \in X} A_M\right) \times \left(\bigcup_{M \in Y} A_{\neg M}\right).$$

We will sometimes omit the superscript and write A_1 or A_2 if the instance I is clear from context. The automaton A_1 accepts all generalizing expressions and the automaton A_2 accepts all non-generalizing expressions:

$$L(A_1) = \{e \in L(G') : \mathsf{solves}(e, I)\} \quad \text{and}$$
$$L(A_2) = \{e \in L(G') : \mathsf{solves}(e, X) \land \neg\mathsf{solves}(e, Y)\}.$$

Let us call these *instance automata*. Our goal is to keep track of how these automata evaluate over the expressions admitted by a grammar G, in order of increasing parse tree depths.

Suppose $A_1 = (Q_1, \Delta, F_1, \delta_1)$ and $A_2 = (Q_2, \Delta, F_2, \delta_2)$. Note that, as constructed, A_1 and A_2 are non-deterministic top-down automata. We will consider recursion tables (described in Section 5.1) whose entries range over the powerset $\mathcal{P}(Q_1 \sqcup Q_2)$. On an input grammar tree, our automaton A_I will keep track of the rows of its corresponding recursion table.

Let us fix a grammar $G = (S, N, \Delta, P)$. To define its recursion table $T(G)$, we order its nonterminals as $N_1, N_2, \ldots, N_k$, with $N_1 = S$. Now let $H_i : \mathcal{P}(Q_1 \sqcup Q_2)^k \to \mathcal{P}(Q_1 \sqcup Q_2)$ be the operator defined by the equation

$$H_i(V) \;=\; \bigcup_{(N_i,\, t) \in P} [\![t]\!]_V^{A_1} \sqcup [\![t]\!]_V^{A_2}, \qquad V \in \mathcal{P}(Q_1 \sqcup Q_2)^k.$$

The notation $[\![t]\!]_V^{A_j}$ denotes the subset of Q_j reachable by running the automaton

$$A_j' = \left(Q_j, \Delta \sqcup \left\{\mathsf{rhs}_{N_s} : N_s \in N\right\}, F_j, \delta_j'\right)$$

on term t, where $\delta_j'(q, \mathsf{rhs}_{N_s}) = \mathsf{true}$ for each $q \in V_s \cap Q_j$ and nonterminal N_s and $\delta_j'(q, x) = \delta_j(q, x)$ for all other $q \in Q_j, x \in \Delta$. The intuition is that H_i computes the states of the instance automata which can be reached by some expression generated by N_i, given an assumption about what states have already been reached.

The operator $H : \mathcal{P}(Q_1 \sqcup Q_2)^k \to \mathcal{P}(Q_1 \sqcup Q_2)^k$ defined by $H(V) = \langle (H_1(V), \ldots, H_k(V)\rangle$ is monotone with respect to component-wise inclusion of sets, and thus the following sequence converges to a fixpoint after $n \leq k(|Q_1| + |Q_2|)$ steps:

$$\langle \emptyset, \ldots, \emptyset \rangle =: Z_0,\, H(Z_0),\, H^2(Z_0),\, \ldots,\, H^n(Z_0) = H^{n+1}(Z_0).$$

We define the recursion table $T(G)$ as follows. There are $k = |N|$ columns and $n^* + 1$ rows, where $n^* := k(|Q_1| + |Q_2|)$. The entry at row i, column j, denoted $T(G)[i, j]$[2], consists of the subset of

[2]For convenience, we index rows starting from zero and columns starting from one.

values from $Q_1 \sqcup Q_2$ that are first achieved at parse tree depth i for nonterminal N_j. For $1 \leq j \leq k$:

$$T(G)[0, j] \coloneqq \emptyset$$
$$T(G)[i, j] \coloneqq H^i(Z_0)_j \setminus H^{i-1}(Z_0)_j \qquad 0 < i \leq n^*.$$

By construction, for the ordering given by

$$\mathsf{depth}(e, G) \leq \mathsf{depth}(e', G),$$

a grammar G solves $I = (X, Y)$ if and only if there is some row i for which $F_1 \cap T(G)[i, 1] \neq \emptyset$ and for all $0 \leq j < i$ we have $F_2 \cap T(G)[j, 1] = \emptyset$. That is, the grammar G solves I if and only if there is some depth i at which it generates a solution for $X \cup Y$ and *all* non-generalizing expressions cannot be generated in depth less than i. Let us say that column 1 of $T(G)$ is *acceptable* if this holds.

We can now define the tree automaton A_I whose language is

$$L(A_I) = \{t \in T_{\Gamma(\Delta,N)} \;:\; \mathsf{solves}(\mathsf{ext}(\mathsf{dec}(t), G'), I)\}.$$

First we summarize the high-level operation of A_I as it relates to recursion tables and then give a detailed construction after.

Summary. On an input $t \in T_{\Gamma(\Delta,N)}$, the automaton guesses the construction of the recursion table $T(t) \coloneqq T(\mathsf{ext}(\mathsf{dec}(t), G'))$ starting from row 0 and working downward to row n^*. As it guesses the rows, the automaton checks the newest row can be produced from preceding rows and that each entry $T(t)[i, j]$ in fact contains $H^i(Z_0)_j \setminus H^{i-1}(Z_0)_j$, i.e., it is the set of all new domain values that can be constructed using previously constructed domain values. To do this, it simulates the instance automata to check that each value in $T(t)[i, j]$ can be generated using some production (N_j, α), with each nonterminal that appears in α interpreted as a value in a previously guessed row. Furthermore, to check that $T(t)[i, j]$ contains *all* new values that can be generated at stage i, the automaton tracks the set of remaining values that have not yet been generated by stage i, namely $(Q_1 \sqcup Q_2) \setminus H^i(Z_0)_j$, and verifies that none of them are generated in stage i.

After each row, the automaton monitors whether column 1 is *acceptable*, as described earlier. Recall this corresponds to checking whether a generalizing or non-generalizing expression is encountered first. If no generalizing expression has been found yet and a non-generalizing one is found at row i, that is $F_2 \cap T(t)[i, 1] \neq \emptyset$ and for all $j \leq i$ we have $F_1 \cap T(t)[j, 1] = \emptyset$, then the automaton rejects. Otherwise, if a generalizing expression is found at row i, that is $F_1 \cap T(t)[i, 1] \neq \emptyset$, then the automaton accepts. Finally, if no generalizing expression is found at row n^*, equivalently $F_1 \cap T(t)[i, 1] = \emptyset$ for all $0 \leq i \leq n^*$, then the automaton rejects.

Construction. We define a two-way alternating tree automaton $A_I = (Q, \Gamma(\Delta, N), I, \delta)$ with acceptance defined by the existence of a finite run satisfying the transition formulas.

Recall k is the number of nonterminals. We use D as a shorthand for the powerset $\mathcal{P}(Q_1 \sqcup Q_2)$, and so possible rows of the recursion table are drawn from D^k. We abuse notation by writing $L \cap L'$ and $L \cup L'$ for componentwise intersection and disjunction over vectors $L, L' \in D^k$. The automaton operates in a few different modes. In **mode 1** it moves to the top of the input tree t. From **mode 1** it enters **mode 2**, in which it guesses which element $C \in D^k$ appears as the next row of $T(t)$. In **mode 3** it traverses the right spine of the tree to verify the guess C. For each nonterminal N_i encountered along the right spine of t, this involves guessing which subset $U \subseteq C_i$ a given production for N_i should reach, given a vector $L \in D^k$ consisting of all previously reached values for all nonterminals. In **mode 4** and **mode 5** it attempts to verify these guesses. In **mode 4**, it simulates the modified instance automata A_1' on the right-hand side of a given production to check that all values in U are reachable, assuming those in L are reachable. In **mode 5**, it simulates A_2' to check that all values in $R_i \coloneqq (Q_1 \sqcup Q_2) \setminus (L \cup C)_i$ are not reachable, again assuming those in L are reachable. Note that after the automaton reaches the end of the productions on the right spine of t, it simulates **modes**

4 and **5** as if the productions P' from the base grammar G' were present in the tree. Finally, it enters **mode 6** to check if the partially guessed column corresponding to the starting nonterminal is already acceptable, and if so it accepts. Otherwise it verifies that no non-generalizing expression has yet been constructed and enters **mode 1** to return to the root of t.

We describe the states Q and their transition formulas grouped by functionality. Below we use $i, j \in [k]$, $m \in \{1,2\}$, $u \in Q_1 \sqcup Q_2$, $u_1 \in Q_1$, $u_2 \in Q_2$, $U, V \in D$, $L, R, R', C, C', W \in D^k$, $N_i, N_j \in N$, $f \in \Delta^r$, and $t_1, \ldots, t_r \in T_\Delta(\{\mathsf{rhs}_{N_i} : N_i \in N\})$. We use an underscore "_" to describe a default transition when no other case matches.

Mode 1. Reset to the top of the input tree. States are drawn from

$$\mathbf{M1} := (D^k)^3 \times \{\mathbf{reset}, \mathbf{start}\}.$$

- $\delta(\langle L, C, R, \mathbf{reset}\rangle, \mathrm{root}) = (\mathbf{down}, \langle L, C, R, \mathbf{start}\rangle)$
- $\delta(\langle L, C, R, \mathbf{reset}\rangle, _) = (\mathbf{up}, \langle L, C, R, \mathbf{reset}\rangle)$
- $\delta(\langle L, C, R, \mathbf{start}\rangle, \mathsf{lhs}_{N_i}) = (\mathbf{stay}, \langle L, C, R, i, \mathbf{row}\rangle)$

Mode 2. Guess next row of the recursion table. States drawn from

$$\mathbf{M2} := (D^k)^3 \times [c] \times \{\mathbf{row}\}.$$

- $\delta(\langle L, C, R, i, \mathbf{row}\rangle, _) =$
  $\bigvee_{(C',R') \in \mathbf{split}(R)} (\mathbf{stay}, \langle L \cup C, C', C', R', i, \mathbf{prod}\rangle)$
  with $\mathbf{split}(R) := \left\{(C', R') \in D^k \times D^k \ : \ C' \cup R' = R,\ C' \neq \{\emptyset\}^k\right\}$

Mode 3. Guess the contributions of productions to each row entry. States drawn from

$$\mathbf{M3} := (D^k)^4 \times [c] \times \{\mathbf{prod}\}.$$

- $\delta(\langle L, C, W, R, i, \mathbf{prod}\rangle, \mathsf{lhs}_{N_j}) =$
  $\bigvee_{\{(U,V) : U \cup V = C_j\}} (\mathbf{left}, \langle L, U, \mathbf{hit}\rangle) \wedge (\mathbf{left}, \langle L, R_j, \mathbf{miss}\rangle) \wedge$
  $(\mathbf{right}, \langle L, \langle C_1, \ldots, C_{j-1}, V, \ldots, C_c\rangle, W, R, i, \mathbf{prod}\rangle)$
- $\delta(\langle L, C, W, R, i, \mathbf{prod}\rangle, \mathrm{end}) =$
  if $\exists (N_j \in N \setminus N').\, C_j \neq \emptyset$
  then false
  else

$$\Big((\mathbf{stay}, \langle L_i \cup W_i, \mathbf{solve}\rangle)$$
$$\vee\Big((\mathbf{stay}, \langle L, W, R, \mathbf{reset}\rangle)) \wedge (\mathbf{stay}, \langle L_i \cup W_i, \mathbf{ok}\rangle)\Big)\Big)$$
$$\wedge\Big(\bigwedge_{N_j \in N'} \bigwedge_{u \in C_j} \bigvee_{(N_j,\alpha) \in P'} (\mathbf{stay}, \langle L, \{u\}, \alpha, \mathbf{hit}\rangle)\Big)$$
$$\wedge\Big(\bigwedge_{N_j \in N'} \bigwedge_{(N_j,\alpha) \in P'} (\mathbf{stay}, \langle L, R_j, \alpha, \mathbf{miss}\rangle)\Big)$$

Mode 4. Check a set of values can be reached. States drawn from

$$\begin{aligned}
\mathbf{M4} &\coloneqq \mathbf{M4a} \cup \mathbf{M4b} \\
\mathbf{M4a} &\coloneqq D^k \times D \times \{\mathbf{hit}\} \cup ((Q_1 \sqcup Q_2) \times (D^k \times \{1,2\})) \\
\mathbf{M4b} &\coloneqq \big(D^k \times D \times \mathit{subterms}(P') \times \{\mathbf{hit}\}\big) \\
&\qquad \cup \big((Q_1 \sqcup Q_2) \times (D^k \times \{1,2\} \times \mathit{subterms}(P'))\big), \\
&\text{where } \mathit{subterms}(P') = \textstyle\bigcup_{(N_i,\alpha) \in P'} \mathit{subterms}(\alpha)
\end{aligned}$$

Transitions for **M4a**:

- $\delta(\langle L, U, \mathbf{hit}\rangle, _) =$ $\bigwedge_{u_1 \in U \cap Q_1}(\mathbf{stay}, \langle u_1, \langle L, 1\rangle\rangle) \wedge \bigwedge_{u_2 \in U \cap Q_2}(\mathbf{stay}, \langle u_2, \langle L, 2\rangle\rangle)$
- $\delta(\langle u, \langle L, m\rangle\rangle, x) = \mathit{adorn}(\langle L, m\rangle, \delta_m(u, y)), \quad x \in \Delta$
- $\delta(\langle u, \langle L, m\rangle\rangle, \mathsf{rhs}_{N_i}) = \mathsf{true} \quad \text{if } u \in L_i$
- $\delta(\langle u, \langle L, m\rangle\rangle, \mathsf{rhs}_{N_i}) = \mathsf{false} \quad \text{if } u \notin L_i$

Transitions for **M4b**:

- $\delta(\langle L, U, \alpha, \mathbf{hit}\rangle, _) =$ $\bigwedge_{u_1 \in U \cap Q_1}(\mathbf{stay}, \langle u_1, \langle L, 1, \alpha\rangle\rangle) \wedge \bigwedge_{u_2 \in U \cap Q_2}(\mathbf{stay}, \langle u_2, \langle L, 2, \alpha\rangle\rangle)$
- $\delta(\langle u, \langle L, m, f(t_1, \ldots, t_r)\rangle\rangle, _) = \mathit{adorn}'(\langle L, m, t_1, \ldots, t_r\rangle, \delta_m(u, f))$
- $\delta(\langle u, \langle L, m, \mathsf{rhs}_{N_i}\rangle\rangle, _) = \mathsf{true} \quad \text{if } u \in L_i$
- $\delta(\langle u, \langle L, m, \mathsf{rhs}_{N_i}\rangle\rangle, _) = \mathsf{false} \quad \text{if } u \notin L_i$

The notation $\mathit{adorn}(s, \varphi)$ represents the transition formula obtained by replacing each atom (i, q) in the Boolean formula φ by the atom $(i, \langle q, s\rangle)$. The notation $\mathit{adorn}'(\langle s, t_1, \ldots, t_r\rangle, \varphi)$ represents the transition formula obtained by replacing each atom (i, q) in φ by the atom $(\mathbf{stay}, \langle q, \langle s, t_i\rangle\rangle)$[3].

Mode 5. Check values cannot be reached. States drawn from

$$\begin{aligned}
\mathbf{M5} &\coloneqq \mathbf{M5a} \cup \mathbf{M5b} \\
\mathbf{M5a} &\coloneqq D^k \times D \times \{\mathbf{miss}\} \cup ((Q_1 \sqcup Q_2) \times (D^k \times \{1,2\} \times \{\bot\})) \\
\mathbf{M5b} &\coloneqq \big(D^k \times D \times \mathit{subterms}(P') \times \{\mathbf{miss}\}\big) \\
&\qquad \cup \big((Q_1 \sqcup Q_2) \times (D^k \times \{1,2\} \times \{\bot\} \times \mathit{subterms}(P'))\big)
\end{aligned}$$

Transitions for **M5a**:

- $\delta(\langle L, U, \mathbf{miss}\rangle, _) =$ $\bigwedge_{u \in U \cap Q_1}(\mathbf{stay}, \langle u, \langle L, 1, \bot\rangle\rangle) \wedge \bigwedge_{u \in U \cap Q_2}(\mathbf{stay}, \langle u, \langle L, 2, \bot\rangle\rangle)$
- $\delta(\langle u, \langle L, m, \bot\rangle\rangle, x) = \mathit{adorn}(\langle L, m, \bot\rangle, \mathit{dual}(\delta_m(u, x))), \quad x \in \Delta$
- $\delta(\langle u, \langle L, m, \bot\rangle\rangle, \mathsf{rhs}_{N_i}) = \mathsf{true} \quad \text{if } u \notin L_i$
- $\delta(\langle u, \langle L, m, \bot\rangle\rangle, \mathsf{rhs}_{N_i}) = \mathsf{false} \quad \text{if } u \in L_i$

Transitions for **M5b**:

- $\delta(\langle L, U, \alpha, \mathbf{miss}\rangle, _) =$ $\bigwedge_{u \in U \cap Q_1}(\mathbf{stay}, \langle u, \langle L, 1, \bot, \alpha\rangle\rangle) \wedge \bigwedge_{u \in U \cap Q_2}(\mathbf{stay}, \langle u, \langle L, 2, \bot, \alpha\rangle\rangle)$

[3]Here we assume directions in δ_1 and δ_2 are the numbers 1(**left**), 2(**right**), 3 . . ., etc.

- $\delta(\langle u, \langle L, m, \bot, f(t_1, \ldots, t_r)\rangle\rangle, _) =$
  $adorn'(\langle L, m, \bot, t_1, \ldots, t_r\rangle, dual(\delta_m(u, f)))$
- $\delta(\langle u, \langle L, m, \bot, \mathsf{rhs}_{N_i}\rangle\rangle, _) = \mathsf{true} \quad \text{if } u \notin L_i$
- $\delta(\langle u, \langle L, m, \bot, \mathsf{rhs}_{N_i}\rangle\rangle, _) = \mathsf{false} \quad \text{if } u \in L_i$

The notation $dual(\varphi)$ represents a transition formula obtained by replacing conjunction with disjunction and vice versa in the (positive) Boolean formula φ.

Mode 6. Check the first column is acceptable or could still be acceptable later. States drawn from

$$\mathbf{M6} := D \times \{\mathbf{solve}, \mathbf{ok}\}.$$

- $\delta(\langle U, \mathbf{solve}\rangle, _) = \text{if } U \cap F_1 \neq \emptyset \text{ then true else false}$
- $\delta(\langle U, \mathbf{ok}\rangle, _) = \text{if } U \cap F_2 \neq \emptyset \text{ then false else true}$

Any transition not described by the rules above has transition formula false. The full set of states for the automaton A_I is $Q := \mathbf{M1} \sqcup \mathbf{M2} \sqcup \mathbf{M3} \sqcup \mathbf{M4} \sqcup \mathbf{M5} \sqcup \mathbf{M6}$, and the initial state set is $I = \{\langle \emptyset, \emptyset, \{Q_1 \sqcup Q_2\}^k, \mathbf{reset}\rangle\}$. By construction, we have the following.

Lemma 5.1. $L(A_I) = \{t \in T_{\Gamma(\Delta,N)} \ : \ \mathsf{solves}(\mathsf{ext}(\mathsf{dec}(t), G'), I)\}$.

Proof. Appendix A.1. □

5.3 Main Theorem

The construction of A_I relied on the assumption that, for each example, we can construct finite tree automata that recognize the sets of all expressions in the base language that are consistent or inconsistent with that example. There is a large class of languages for which this assumption holds, a class which was identified in [Krogmeier and Madhusudan 2022, 2023] and called FAC languages. Our proof thus yields decidability of (depth order) DSL synthesis for all FAC base languages at once, including regular expressions on finite words, propositional modal logic on finite Kripke structures, CTL on finite Kripke structures, LTL over periodic words, context-free grammars on finite words, first-order queries over the ordered rational numbers, and the DSL from Gulwani's work on spreadsheet programs [Gulwani 2011], in addition to several finite-variable logics.

Our main theorem is the following:

Theorem 5.2. *The DSL synthesis problem is decidable for FAC languages with expressions ordered by parse tree depth. Furthermore, for such languages the set of solutions corresponds to a regular set of trees.*

Proof. Given meta-grammar $\mathcal{G}$ and instances $I_1, \ldots, I_l$, we construct the product

$$A := A(\mathcal{G}) \times \mathsf{convert}\big(\textstyle\bigtimes_{i\in[l]} A_{I_i}\big),$$

where $\mathsf{convert}(B)$ is a procedure for converting a two-way alternating tree automaton B to a top-down non-deterministic automaton in time $\exp(|B|)$, as explained in [Cachat 2002; Vardi 1998]. We have by construction and Lemma 5.1 that

$$L(A) = \big\{\, t \in L(\mathcal{G}) \ : \ \textstyle\bigwedge_{i\in[l]} \mathsf{solves}(\mathsf{ext}(\mathsf{dec}(t), G'), I_i) \,\big\}.$$

Existence of solutions is decided by an automaton emptiness procedure which runs in time $\mathrm{poly}(|A|)$, and solutions can be synthesized by outputting $\mathrm{dec}(t)$ for any $t \in L(A)$ in the same time. □

Corollary 5.3. *DSL synthesis is decidable in time*

$$\mathrm{poly}(|\mathcal{G}|) \cdot \exp(l \cdot \exp(m))),$$

where l is the number of few-shot learning instances and m is the maximum size over all instance automata.

We note that, for an instance $I = (X, Y)$, the instance automata size grows exponentially in the number of examples $|X| + |Y|$.

6 DECIDABLE GRAMMAR SYNTHESIS

In this section, we show decidability of a natural relaxation of DSL synthesis, which we call *grammar synthesis*. Given a set of few-shot learning instances $X_1, \ldots, X_l$, along with a meta-grammar constraint $\mathcal{G}$, the requirement is to synthesize a grammar G satisfying $\mathcal{G}$ such that each X_j has a solution in G. There is no longer a constraint on the order of expressions and thus the learning instances are sets of examples, with no separation into training and testing sets. This enables a simpler decision procedure.

6.1 Grammar Synthesis Problem

The grammar synthesis problem is defined as follows.

Problem (Grammar Synthesis)**.**
Parameters:

- Finite set of nonterminals N
- Base language with $G' = (S', N', \Delta, P')$ and $N' \subseteq N$

Input:

- Example sets $X_1, \ldots, X_l$
- Meta-grammar $\mathcal{G}$ over $\Gamma(\Delta, N)$

Output: A grammar $G = (S, N, \Delta, P)$ such that:

(1) for each j, there exists $e \in \mathrm{ext}(G, G')$ that solves X_j
(2) it satisfies constraints imposed by $\mathcal{G}$, i.e., $\mathrm{enc}(G) \in L(\mathcal{G})$

We write solves(G, X) if Item 1 holds for a grammar G and set of examples X.

6.2 Automaton Construction

We now establish decidability of the grammar synthesis problem. The decision procedure has the same high-level structure as in Section 5. We construct tree automata A_X that read grammar trees t as input and accept if solves$(\mathrm{dec}(t), X)$ holds. The product automaton over all example sets X_j gives us an automaton from which we can extract a grammar that solves all sets. First we summarize the operation of A_X and then give a more detailed construction after.

Summary. The automaton A_X reads a grammar tree over alphabet $\Gamma(\Delta, N)$, as before. But now it no longer needs to guess and verify the recursion table for the grammar, given that grammar synthesis merely requires the existence of at least one solution for X. Similar to A_I from Section 5.2, the automaton A_X explores potential solutions by simulating an automaton A_1, here defined as

$$A_1 \coloneqq \bigtimes_{M \in X} A_M, \quad \text{with } L(A_1) = \{e \in L(G') \,:\, \mathsf{solves}(e, X)\},$$

which accepts all expressions in the base language that solve X, and where for each $M \in X$, A_M is an example automaton as described in Section 5.2. Unlike the previous construction of A_I, A_X does

not need to keep track of the depths at which various properties are achieved by expressions in the grammar. This simplifies its description considerably.

Intuitively, the automaton operates by walking up and down the input grammar tree to guess a parse tree for an expression e that solves X. When it reads the right-hand side of a production, it simulates A_1, stopping with acceptance if it completes a parse tree branch on which A_1 satisfies its transition formula. Otherwise it rejects if A_1 is not satisfied, or it continues guessing the construction of a parse tree if it reads a nonterminal symbol. Each time it reads a nonterminal, it must guess which of the productions for that nonterminal should be used. If any sequence of such guesses and simulations of A_1 leads to a completed parse tree which satisfies the transition formulae for A_1, then the existence of a solution in the grammar is guaranteed, and vice versa.

Construction. Suppose $A_1 = (Q_1, \Delta, I_1, \delta_1)$. We define a two-way alternating tree automaton $A_X = (Q, \Gamma(\Delta, N), I, \delta)$. The automaton operates in two modes. In **mode 1**, it walks to the top spine of the input tree in search of productions for a specific nonterminal. Having found a production, it enters **mode 2**, in which it moves down into the term corresponding to the right-hand side of the production, simulating A_1 as it goes.

Below we use $N_i, N_j \in N$, $q \in Q_1$, $x, f \in \Delta$, and $t_1, \ldots, t_r \in T_\Delta(\{\mathsf{rhs}_{N_i} : N_i \in N\})$. Again, we use an underscore "_" to describe a default transition when no other case matches.

Mode 1. Find productions. States are drawn from

$$\mathbf{M1} \coloneqq \big(Q_1 \times \{\mathbf{start}\}\big) \cup \big(Q_1 \times N\big).$$

- $\delta(\langle q, \mathbf{start}\rangle, \mathrm{root}) = (\mathbf{down}, \langle q, \mathbf{start}\rangle)$
- $\delta(\langle q, \mathbf{start}\rangle, \mathsf{lhs}_{N_i}) = (\mathbf{stay}, \langle q, N_i\rangle)$
- $\delta(\langle q, N_i\rangle, \mathsf{lhs}_{N_i}) = (\mathbf{up}, \langle q, N_i\rangle) \vee (\mathbf{left}, q)$
  $\vee(\mathbf{right}, \langle q, N_i\rangle) \vee \big(\bigvee_{(N_i,\alpha)\in P'}(\mathbf{stay}, \langle q, \alpha\rangle)\big)$
- $\delta(\langle q, N_i\rangle, \mathsf{lhs}_{N_j}) = (\mathbf{up}, \langle q, N_i\rangle) \vee (\mathbf{right}, \langle q, N_i\rangle), \quad N_i \neq N_j$
- $\delta(\langle q, N_i\rangle, _) = (\mathbf{up}, \langle q, N_i\rangle) \vee \big(\bigvee_{(N_i,\alpha)\in P'}(\mathbf{stay}, \langle q, \alpha\rangle)\big)$

Mode 2. Read productions. States drawn from

$$\mathbf{M2} \coloneqq Q_1 \cup \big(Q_1 \times \mathit{subterms}(P')\big),$$
$$\text{where } \mathit{subterms}(P') = \textstyle\bigcup_{(N_i,\alpha)\,\in P'} \mathit{subterms}(\alpha).$$

- $\delta(q, x) = \delta_1(q, x)$
- $\delta(q, \mathsf{rhs}_{N_i}) = (\mathbf{stay}, \langle q, N_i\rangle)$
- $\delta(\langle q, f(t_1, \ldots, t_r)\rangle, _) = \mathit{adorn}(t_1, \ldots, t_r, \delta_1(q, f))$
- $\delta(\langle q, \mathsf{rhs}_{N_i}\rangle, _) = (\mathbf{stay}, \langle q, N_i\rangle) \vee \big(\bigvee_{(N_i,\alpha)\in P'}(\mathbf{stay}, \langle q, \alpha\rangle)\big)$

The notation $\mathit{adorn}(t_1, \ldots, t_r, \varphi)$ represents a transition formula obtained by replacing each atom of the form (i, q) in the Boolean formula φ by the atom $(\mathbf{stay}, \langle q, t_i\rangle)$[4].

Any transition not described by the rules above has transition formula false. The full set of states for the automaton A_X is $Q \coloneqq \mathbf{M1} \sqcup \mathbf{M2}$, and the initial state set is $I = \{\langle q, \mathbf{start}\rangle : q \in I_1\} \subseteq \mathbf{M1}$. We have the following by construction.

Lemma 6.1. $L(A_X) = \big\{t \in T_{\Gamma(\Delta,N)} : \mathsf{solves}(\mathsf{ext}(\mathsf{dec}(t), G'), X)\big\}$.

Proof. Appendix B.1. □

[4]Again, we assume directions in δ_1 are the numbers $1(\mathbf{left})$, $2(\mathbf{right})$, $3\ldots$, etc.

We use this to prove the following theorem:

Theorem 6.2. *The grammar synthesis problem is decidable for FAC languages. Furthermore, the set of solutions corresponds to a regular set of trees.*

Proof. Given meta-grammar $\mathcal{G}$ and instances $X_1, \ldots, X_l$, we construct the product

$$A := A(\mathcal{G}) \times \mathsf{convert}\big(\bigtimes_{i \in [l]} A_{X_i}\big)^5.$$

By construction and Lemma 6.1 we have

$$L(A) = \big\{t \in L(\mathcal{G}) \;:\; \bigwedge_{i \in [l]} \mathsf{solves}(\mathsf{ext}(\mathsf{dec}(t), G'), X_i)\big\}.$$

Existence of solutions is decided by an automaton emptiness procedure which runs in time $\mathrm{poly}(|A|)$, and solutions can be synthesized by outputting $\mathrm{dec}(t)$ for any $t \in L(A)$ in the same time. □

Corollary 6.3. *Grammar synthesis is decidable in time*

$$\mathrm{poly}(|\mathcal{G}|) \cdot \exp(l \cdot m)),$$

where l is the number of instances and m is the maximum size over all instance automata A_X.

6.3 Remark on constructions

The ideas behind the constructions in Sections 5 and 6 can be used to prove the following more general statements. In the case of Section 6, we have the following.

Lemma 6.4. *Given a tree automaton A, there is a tree automaton B_A that accepts an encoding of a grammar G if and only if $L(A) \cap L(G) \neq \emptyset$.*

In the context of Section 6, the automaton B_A corresponds to A_X and the automaton A corresponds to the automaton A_1.

The more complex construction from Section 5 corresponds to the following.

Lemma 6.5. *Given tree automata A and B, there is a tree automaton C that accepts an encoding of a grammar G if and only if there is some $i \in \mathbb{N}$ such that $L(A) \cap L(G)_i \neq \emptyset$ and $L(B) \cap L(G)_i = \emptyset$, where $L(G)_i$ is the set of terms obtained at iteration i of the fixpoint computation for $L(G)$.*

In this case, the automata A and B correspond to A_1^I and A_2^I.

7 RELATED WORK

Grammar induction. There is a large body of work on the synthesis of grammar/automata representations of formal languages, e.g., the well-known L* [Angluin 1987] and RPNI [Oncina and García 1992] algorithms for learning representations of regular languages in terms of DFAs. Vanlehn and Ball [Vanlehn and Ball 1987] explored an approach to context-free grammar induction based on version space algebra [Mitchell 1982]. The use of tree automata in our decidability proofs is related to version space algebra, the main point of overlap being the idea to consider equivalence classes of grammars and expressions modulo examples.

Applications of grammar induction to learning valid program inputs, fuzzing, test generation, and learning context-free grammars that match labeled examples of different data formats have spurred recent research in this vein [Bastani et al. 2017; Kulkarni et al. 2021; Miltner et al. 2023]. These are very different problems than what we consider, as the specification is driven by the syntax of examples, i.e., the structure of finite words; in contrast, the properties of grammars relevant to this paper are driven by language semantics and its relationship with input example data. And most

[5] The convert procedure is the same as that in Section 5.3.

importantly, we look for grammars with specific biases, which are specified by few-shot learning probelms split into training and testing sets.

Program synthesis. The significance of grammar and inductive bias in program synthesis is well recognized. The tradeoff between expressive grammars and synthesizer performance was studied for SyGuS problems in [Padhi et al. 2019]. The well-known case of FlashFill [Gulwani 2011] was able to scale program synthesis from examples to practial use cases in Microsoft Excel. One of the major reasons for success was a carefully crafted DSL that successfully navigated a tradeoff between expressivity and tractable search. Practical implementations of DSL synthesis could enable easier design of synthesizers for new programming-by-example domains.

Library learning. Recent work explores the problem of compressing a given corpus of programs by finding commonly occurring programming patterns and abstracting them as functions [Bowers et al. 2023; Cao et al. 2023]. In each of these, the goal is to find a set of programs which can be composed to generate the input corpus, but which also serves to compress it. Hence the problem they study is quite different, as there is no notion of learning from examples. Closer in spirit to our work is DreamCoder [Ellis et al. 2023], which solves few-shot learning instances by iteratively growing a library of programs that capture patterns used in previous solutions. Main differences with our work are that DreamCoder does not give any formal guarantee about how well learned abstractions generalize. It also does not produce grammars, which is a subtle difference in the context of our problem. Intuitively, a grammar can be thought of as a library of programs organized into equivalence classes, where programs must compose with each other in ways that respect the classes.

Applications of tree automata. Tree automata underlie several deep results on synthesis of finite-state systems, e.g., the solutions to Church's problem [Church 1963] by Büchi and Landweber [Buchi and Landweber 1969] and Rabin [Rabin 1969]. A different use of tree automata gives the foundation for fixed-parameter tractable algorithms for model checking on parameterized classes of graphs [Courcelle and Engelfriet 2012; Habel 1992]. Our use of tree automata is quite different than the latter use, which uses automata to read in decompositions of graphs in order to check whether they satisfy a fixed property in a logic. The tree automata in this paper instead read parse trees of formulas or programs in order to check whether they are satisfied by a fixed example structure, and there is no restriction of the examples to efficient classes of structures. We simply need that given an arbitrary finitely-presented structure, we can construct a tree automaton that reads parse trees and evaluates them over the structure. This idea was used recently to prove decidability results for learning in finite-variable logics [Krogmeier and Madhusudan 2022] and several other symbolic languages [Krogmeier and Madhusudan 2023], and the decidability results of this paper apply to DSL synthesis over all languages studied there. The idea has also been used recently for decidability results in program synthesis, e.g., synthesizing reactive programs from formal specifications [Madhusudan 2011] and uninterpreted programs from sketches [Krogmeier et al. 2020], as well as practical algorithmic foundations for program synthesis, e.g., [Gulwani 2011; Polozov and Gulwani 2015; Wang et al. 2017b, 2018b].

Synthesizing abstractions for program synthesis. The idea of using abstract domains and transformers for program synthesis can help make search more tractable when suitable abstractions can be designed manually, e.g., [Guria et al. 2023; Wang et al. 2017a,b], but often they are hard to design. Some recent work has explored synthesizing the abstract domain and transformers automatically for programming-by-example problems [Wang et al. 2018a].

8 CONCLUSION

In this work we studied the problem of synthesizing DSLs for few-shot learning in symbolic domains. We presented a novel formulation of DSL synthesis that uses a *learning* specification, where the

goal is to learn a DSL given some instances of few-shot learning problems as input. The learned DSL is defined using a grammar, which itself is defined over a base language whose semantics is used to give semantics to the DSL. Each few-shot learning instance in the input is split into training and testing datasets, which indicate an expectation about the kinds of properties a useful DSL should be capable of expressing using formulas or programs of small complexity. The notion of complexity is formalized in terms of a fixed ordering $<$ on expressions which can be computed given a DSL and two expressions in the language. The problem asks for a grammar which is expressive enough to contain solutions for each few-shot learning instance in the input, while ensuring that for each one, the smallest expressions according to $<$ that are consistent with the training set are also consistent with the testing set. In cases where expressions can easily be enumerated in increasing order, e.g., parse tree size or depth, a DSL satsifying the problem specification can be used to solve few-shot learning problems by enumerating small expressions and picking the first one that is consistent with the training dataset. Intuitively, the DSL synthesis problem asks for a DSL such that Occam's razor, i.e., picking the simplest concept consistent with input (training) data, results in expressions which generalize well, in the sense that they are also consistent with testing data.

We proved that DSL synthesis is decidable when expressions are ordered by parse tree depth for any base language whose semantics can be evaluated by a finite tree automaton. Specifically, what this means is that for any fixed example, a tree automaton can be constructed that evaluates the semantics of arbitrarily large expressions in the base language. This class of base languages includes several natural ones from the literature, including regular expressions on finite words, propositional modal logic and computation tree logic on finite Kripke structures, linear temporal logic over periodic words, context-free grammars on finite words, first-order queries over the ordered rational numbers, and the language from Gulwani's work on program synthesis for spreadsheets [Gulwani 2011], in addition to several finite-variable logics.

The proof of decidability relied on constructing tree automata which read trees that encode grammars and verify that solutions to the input learning instances exist in the grammar, and furthermore, that expressions which are consistent only with the training dataset are not easy to express. Existence of solutions to DSL synthesis was reduced to tree automaton non-emptiness, and thus smallest grammars solving the problem can be constructed using standard tree automaton emptiness checking algorithms.

We also studied the grammar synthesis problem, a relaxation of DSL synthesis which removes the ordering requirement and simply asks for a DSL that contains solutions for each few-shot learning problem presented in the input. We proved decidability in this setting as well and obtained a more efficient decision procedure.

In future work, we plan to explore practical implementations for solving DSL synthesis. It will be interesting to explore whether practical DSL synthesis can indeed be realized using few-shot learning problems as specifications, and if so, how much data is needed to arrive at useful DSLs in specific domains.

ACKNOWLEDGMENTS

REFERENCES

Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and Computation* 75, 2 (1987), 87–106. https://doi.org/10.1016/0890-5401(87)90052-6

Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing Program Input Grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 95–110. https://doi.org/10.1145/3062341.3062349

Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. 2023. Top-Down Synthesis for Library Learning. *Proc. ACM Program. Lang.* 7, POPL, Article 41 (jan 2023), 32 pages. https://doi.org/10.1145/3571234

J. Richard Buchi and Lawrence H. Landweber. 1969. Solving Sequential Conditions by Finite-State Strategies. *Trans. Amer. Math. Soc.* 138 (1969), 295–311. http://www.jstor.org/stable/1994916

Thierry Cachat. 2002. *Two-Way Tree Automata Solving Pushdown Games.* Springer-Verlag, Berlin, Heidelberg, 303–317.

David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. 2023. Babble: Learning Better Abstractions with E-Graphs and Anti-Unification. *Proc. ACM Program. Lang.* 7, POPL, Article 14 (jan 2023), 29 pages. https://doi.org/10.1145/3571207

Alonzo Church. 1963. Application of Recursive Arithmetic to the Problem of Circuit Synthesis. *Journal of Symbolic Logic* 28, 4 (1963), 289–290. https://doi.org/10.2307/2271310

H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. 2007. Tree Automata Techniques and Applications. Available on: `http://www.grappa.univ-lille3.fr/tata`. release October, 12th 2007.

Professor Bruno Courcelle and Dr Joost Engelfriet. 2012. *Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach* (1st ed.). Cambridge University Press, New York, NY, USA.

Kevin Ellis, Lionel Wong, Maxwell Nye, Mathias Sablé-Meyer, Luc Cary, Lore Anaya Pozo, Luke Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2023. DreamCoder: growing generalizable, interpretable knowledge with wake–sleep Bayesian program learning. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 381, 2251 (2023), 20220050. https://doi.org/10.1098/rsta.2022.0050

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

Sankha Narayan Guria, Jeffrey S. Foster, and David Van Horn. 2023. Absynthe: Abstract Interpretation-Guided Synthesis. *Proc. ACM Program. Lang.* 7, PLDI, Article 171 (jun 2023), 24 pages. https://doi.org/10.1145/3591285

Annegret Habel. 1992. *Graph-theoretic aspects of HRL's.* Springer Berlin Heidelberg, Berlin, Heidelberg, 117–144. https://doi.org/10.1007/BFb0013882

Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas Reps. 2021. Semantics-Guided Synthesis. *Proc. ACM Program. Lang.* 5, POPL, Article 30 (jan 2021), 32 pages. https://doi.org/10.1145/3434311

Paul Krogmeier and P. Madhusudan. 2022. Learning formulas in finite variable logics. *Proc. ACM Program. Lang.* 6, POPL, Article 10 (jan 2022), 28 pages. https://doi.org/10.1145/3498671

Paul Krogmeier and P. Madhusudan. 2023. Languages with Decidable Learning: A Meta-Theorem. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 80 (apr 2023), 29 pages. https://doi.org/10.1145/3586032

Paul Krogmeier, Umang Mathur, Adithya Murali, P. Madhusudan, and Mahesh Viswanathan. 2020. Decidable Synthesis of Programs with Uninterpreted Functions. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 634–657.

Neil Kulkarni, Caroline Lemieux, and Koushik Sen. 2021. Learning Highly Recursive Input Grammars. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 456–467. https://doi.org/10.1109/ASE51524.2021.9678879

Leslie Lamport. 1983. What Good Is Temporal Logic? *Information Processing 83, R. E. A. Mason, ed., Elsevier Publishers* 83 (May 1983), 657–668. https://www.microsoft.com/en-us/research/publication/good-temporal-logic/

P. Madhusudan. 2011. Synthesizing Reactive Programs. In *Computer Science Logic (CSL'11) - 25th International Workshop/20th Annual Conference of the EACSL (Leibniz International Proceedings in Informatics (LIPIcs))*, Marc Bezem (Ed.), Vol. 12. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 428–442. https://doi.org/10.4230/LIPIcs.CSL.2011.428

Anders Miltner, Devon Loehr, Arnold Mong, Kathleen Fisher, and David Walker. 2023. Saggitarius: A DSL for Specifying Grammatical Domains. arXiv:cs.PL/2308.12329

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

J. Oncina and P. García. 1992. *INFERRING REGULAR LANGUAGES IN POLYNOMIAL UPDATED TIME.* 49–61. https://doi.org/10.1142/9789812797902_0004

Saswat Padhi, Todd Millstein, Aditya Nori, and Rahul Sharma. 2019. Overfitting in Synthesis: Theory and Practice. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 315–334.

Doron Peled and Thomas Wilke. 1997. Stutter-Invariant Temporal Properties Are Expressible without the next-Time Operator. *Inf. Process. Lett.* 63, 5 (sep 1997), 243–246. https://doi.org/10.1016/S0020-0190(97)00133-6

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 (OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 107–126. https://doi.org/10.1145/2814270.2814310

Michael O. Rabin. 1969. Decidability of Second-Order Theories and Automata on Infinite Trees. *Trans. Amer. Math. Soc.* 141 (1969), 1–35. http://www.jstor.org/stable/1995086

Kurt Vanlehn and William Ball. 1987. A Version Space Approach to Learning Context-free Grammars. *Machine Learning* 2, 1 (01 Mar 1987), 39–74. https://doi.org/10.1023/A:1022812926936

Moshe Y. Vardi. 1998. Reasoning about the past with two-way automata. In *Automata, Languages and Programming*, Kim G. Larsen, Sven Skyum, and Glynn Winskel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 628–641.

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017a. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 452–466. https://doi.org/10.1145/3062341.3062365

Xinyu Wang, Greg Anderson, Isil Dillig, and K. L. McMillan. 2018a. Learning Abstractions for Program Synthesis. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 407–426.

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017b. Program synthesis using abstraction refinement. *Proc. ACM Program. Lang.* 2, POPL, Article 63 (dec 2017), 30 pages. https://doi.org/10.1145/3158151

Yuepeng Wang, Xinyu Wang, and Isil Dillig. 2018b. Relational Program Synthesis. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 155 (Oct. 2018), 27 pages. https://doi.org/10.1145/3276525