

Synthesizing DSLs for Few-Shot Learning

PAUL KROGMEIER, University of Illinois at Urbana-Champaign, USA

P. MADHUSUDAN, University of Illinois at 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 samples, the DSL synthesis problem asks for a grammar over the base language that guarantees that small expressions solving training samples also solve corresponding testing samples. We prove that the problem is decidable for a class of languages whose semantics over fixed structures 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 results for variants of the problem where DSLs are only required to express solutions for input learning problems and where DSLs are defined using macro grammars.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; • **Theory of computation** → **Tree languages**; • **Computing methodologies** → *Machine learning*.

Additional Key Words and Phrases: domain-specific language, symbolic learning, program synthesis, tree automata

ACM Reference Format:

Paul Krogmeier and P. Madhusudan. 2025. Synthesizing DSLs for Few-Shot Learning. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 295 (October 2025), 28 pages. <https://doi.org/10.1145/3763073>

1 Introduction

In this work, we are interested in *few-shot learning of symbolic expressions*—e.g., learning classifiers expressed in logic which separate a sample of positive and negative structures or learning programs that compute a function which is consistent with a few input-output examples.

When considering a large class of concepts \mathcal{C} , it is typically impossible to identify a target concept $c \in \mathcal{C}$ from only a small sample S of instances and hence succeed at few-shot learning, as there can be many concepts consistent with the sample. In practice, few-shot symbolic learning, such as program synthesis from examples or learning invariants for programs, etc., is successful because engineers identify a much smaller class of concepts \mathcal{H} —the *hypothesis class*—and learning proceeds over \mathcal{H} . Note that identification of \mathcal{H} may not be necessary with a large amount of data, as in mainstream machine learning. The hypothesis class in *symbolic* learning is defined using a *language* of symbolic expressions pertinent to a specific problem domain—such a language is often referred to as a domain-specific language (DSL)—which captures *typical concepts* that are *useful* in that domain. For *few-shot* learning, there is often also an *ordering* of concepts in \mathcal{H} , and learning algorithms return—appealing to Occam’s razor—the smallest concept in \mathcal{H} with respect to the order that is consistent with the sample S . Typical concept orderings on symbolic expressions are syntactic length or parse tree depth.

Authors’ Contact Information: Paul Krogmeier, University of Illinois at Urbana-Champaign, Urbana, USA, paulmk2@illinois.edu; P. Madhusudan, University of Illinois at Urbana-Champaign, Urbana, USA, madhu@illinois.edu.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2025 Copyright held by the owner/author(s).

ACM 2475-1421/2025/10-ART295

<https://doi.org/10.1145/3763073>

Program synthesis from examples, especially, is replete with hand-engineered DSLs for specific applications. For instance, in spreadsheet automation [Gulwani 2011; Polozov and Gulwani 2015]—where the goal is to complete the columns of a spreadsheet in a manner consistent with a small number of filled-in spreadsheet cells—a suitable DSL identifies common string-manipulation functions that occur in realistic spreadsheet programs [Gulwani 2011]. The SyGuS format (syntax-guided synthesis) for program synthesis makes explicit the discipline of defining hypothesis classes by using *grammars* to define DSLs which syntactically constrain the expressions considered during synthesis [Alur et al. 2015], and recent work explores the case where DSL semantics can be specified in addition to syntax [Kim et al. 2021].

1.1 Formulation of DSL Synthesis

We consider the problem of automatically synthesizing DSLs for few-shot symbolic learning. By solving this problem, few-shot learning in novel domains can be automated further, with learning algorithms working over hypothesis classes defined by synthesized DSLs. The first contribution of this work is a *definition* of the DSL synthesis problem for few-shot learning. We ask:

What formulation of DSL synthesis facilitates few-shot learning?

We formulate DSL synthesis itself as a *learning* problem. Given an application domain D , the idea is to synthesize a DSL given *instances of learning problems* from D .

Consider a set \mathcal{I} of instances of the few-shot learning problem for domain D . Let us assume a “base language” whose syntax is given by a grammar G over a finite signature providing function, relation, and constant symbols, and where expressions of G have fixed semantics. We would like to learn a DSL \mathcal{H} —with *syntax* and *semantics* for expressions of \mathcal{H} formalized using the base language G —that can be used to effectively solve 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 \mathcal{H} to solve each problem $p \in \mathcal{I}$ in the following sense: the *smallest* expressions in \mathcal{H} , measured according to a fixed ordering on expressions, that are consistent with training examples X_p must also be consistent with testing examples Y_p .

This formulation hence asks that a learning bias for D be encoded in the DSL. Note that a learning algorithm that picks the smallest consistent expressions in \mathcal{H} will in fact solve all the learning instances from which \mathcal{H} was learned. In addition to the requirements above, the synthesized DSL must satisfy a “meta-grammar” constraint \mathcal{G} —given in the input—which constrains the kinds of syntax and semantics it may use from the base language.

A DSL \mathcal{H} must satisfy three properties to solve the DSL synthesis problem:

Property 1. For each instance $p \in \mathcal{I}$, \mathcal{H} must express some concept c that solves p , in the sense that c is consistent with both the training and testing examples X_p and Y_p .

Property 2. For each instance $p \in \mathcal{I}$, consider the smallest expressions $e \in \mathcal{H}$, according to the fixed ordering, that express concepts c consistent with the training examples X_p . These concepts c must also be consistent with the testing examples Y_p .

Property 3. The definition of \mathcal{H} using the base language must satisfy constraints \mathcal{G} .

Intuitively, the second property demands that, for any concept c that is expressible in the base language and solves the training set X_p but does not solve the testing set Y_p for some $p \in \mathcal{I}$, the DSL \mathcal{H} must either (1) *not express* c or else (2) ensure that, if c is expressible, then there is another concept expressible using a smaller expression—with respect to the fixed expression ordering—that solves the training and testing sets X_p and Y_p . This property introduces a new learning signal for DSL synthesis which is not part of existing related problems such as library learning [Bowers et al. 2023; Cao et al. 2023] or the problem addressed by systems like DREAMCODER [Ellis et al.

2023], where the goal is to refactor an existing DSL to favor useful concepts. The new signal is about *inexpressivity*, and asks that some concepts should not be expressible, or, if they are, they should not be expressible too succinctly. We wish to find languages specific to domains, and these may be *less expressive* than the base languages we use to define them.

1.2 DSL Synthesis Problems

We study multiple classes of DSL synthesis problems which are quite general in the sense they are not tied to particular logics, programming languages, or underlying theories.

We introduce the four DSL synthesis problems listed below in increasing order of difficulty. In all problems, there is a fixed base language G , and we are given a set of few-shot learning instances \mathcal{I} and a meta-grammar constraint \mathcal{G} . Let us fix an ordering on expressions.

Problem 1: Adequate DSL Synthesis. Is there a DSL \mathcal{H} satisfying constraints \mathcal{G} such that, for every instance $p \in \mathcal{I}$, there is at least one concept expressible in \mathcal{H} that is consistent with the training and testing sets X_p and Y_p ? If so, synthesize the DSL.

Problem 2: Adequate DSL Synthesis with Macros. The same question above, except posed for DSLs defined using grammars with macros.

Problem 3: DSL Synthesis. Is there a DSL \mathcal{H} satisfying constraints \mathcal{G} such that, for every instance $p \in \mathcal{I}$, there are concepts expressible in \mathcal{H} that are consistent with the training set X_p , and the ones expressible most succinctly according to the expression ordering are also consistent with the testing set Y_p . If so, synthesize the DSL.

Problem 4: DSL Synthesis with Macros. The same question above, except posed for DSLs defined using grammars with macros.

Adequate DSL Synthesis captures **Properties 1** and **3** above. It asks whether there is *any* DSL that expresses a solution for each input learning instance. It is equivalent to *DSL Synthesis* where the testing sets are empty, and is therefore independent of the expression ordering. *DSL Synthesis* incorporates **Property 2** and is the problem we have articulated thus far.

It turns out that standard context-free grammars are not powerful enough to capture classes of DSLs that use *macros with parameters*. Consider a language that allows a macro $f(x_1, x_2)$, defined by an expression e with free variables x_1 and x_2 , where $f(e_1, e_2)$ results in the uniform substitution of e_i for x_i in e . Context-free grammars cannot capture such languages when the terms substituted are arbitrarily large. We hence also study the *Adequate DSL Synthesis* and *DSL Synthesis* problems in the setting where DSLs are defined using more expressive macro grammars [Fischer 1968].

1.3 Decidability Results

DSL synthesis is a meta-synthesis problem, i.e., it involves synthesizing a DSL that in turn can solve a set of few-shot synthesis problems, and it is algorithmically very complex. It is a natural question whether there are powerful subclasses where the problem is decidable. More precisely, for a fixed base language with grammar G , using which the hypothesis class \mathcal{H} is defined, we would like algorithms that, when given a set of few-shot learning instances \mathcal{I} and syntax constraint \mathcal{G} , either synthesize \mathcal{H} that solves the instances and satisfies \mathcal{G} or report that no such DSL exists. We allow the semantics of DSLs, defined in terms of G , to be of *arbitrary length*, which makes decidability nontrivial.

Our second contribution is to show that the four problems identified above are *constructively decidable* for a large class of base languages. In particular, we prove that each variant of the DSL synthesis problem is decidable for a class of base languages recently shown to have decidable learning problems [Krogmeier and Madhusudan 2022, 2023]—those for which a specialized recursive

program can evaluate the semantics of arbitrarily large expressions using an amount of memory depending only on a fixed structure over which evaluation occurs.

Our techniques to establish decidability rely 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 automata constructions are significantly more complex than those for learning expressions [Krogmeier and Madhusudan 2022, 2023], both conceptually and in terms of time complexity, and we assume these previous constructions to design tree automata for DSL synthesis.

We point out two factors that make our algorithms complex. First, we design tree automata which read trees that encode DSLs, and these automata must verify the existence of arbitrarily large solutions expressed in these DSLs. Witnessing the non-existence of solutions involves, in general, examining all expression derivations within an encoded DSL. Second, we must check the existence of *minimal* expressions that witness the solvability of each given learning instance. In particular, we need to use alternating quantification on trees to capture the fact that there *exists* a solution e for each instance such that *all* other smaller expressions e' do not solve the training set. We cannot nondeterministically guess e and e' separately, as they are related. We avoid this guessing of e and all smaller e' by essentially leveraging a *dynamic programming algorithm* for evaluating all expressions derivable in a given DSL, in the order of parse tree depth. If we execute this algorithm, then it will check whether the first depth d at which there exists an expression that solves the training examples is the same depth at which an expression first solves both the training *and* testing examples. However, we cannot run the algorithm as we do not have a DSL. But, it turns out that for *any* DSL of arbitrary size, the table of results computed by this dynamic programming algorithm is essentially finite, given a set of learning instances. The contents of cells in the table come from a finite domain for any fixed instance, and thus the table rows repeat at a certain point. Consequently, we can simulate the dynamic programming algorithm using a tree automaton that computes the table for arbitrarily large DSLs encoded as trees.

Contributions. We make the following contributions:

- A novel formulation of DSL synthesis from few-shot learning instances, which asks for a hypothesis class that supports few-shot learning in a domain.
- Decidability results for variants of DSL synthesis over a powerful class of base languages. To the best of our knowledge, these are the first decidability results for DSL synthesis.

The paper is organized as follows. In Section 2, we explore DSL synthesis problems with some examples and applications. In Section 3, we establish some definitions and review background. In Section 4, we formulate various aspects of DSL synthesis. In Sections 5 and 6, we introduce the *Ad-equate DSL Synthesis* and *DSL Synthesis* problems and prove decidability results 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 7, we introduce variants of the previous problems that use macro grammars and prove decidability results. Section 8 reviews related work and Section 9 concludes. Omitted details throughout the paper can be found in the version with appendix [Krogmeier and Madhusudan 2025].

2 DSL Synthesis: Motivation and Examples

In this section, we motivate the DSL synthesis problem with applications to few-shot symbolic learning and synthesis, and explain aspects of our problem formulation.

Computer science is teeming with *symbolic languages* that have been designed by researchers or engineers, not necessarily to be highly expressive but, rather, to be well adapted to specific domains. Such DSLs express *common* properties of the domain using *succinct* expressions and disallow or make more complex the representation of concepts that are irrelevant in the domain.

As explained in Section 1, we aim to identify DSLs that facilitate *few-shot learning*. More precisely, we aim to find DSLs that can succinctly express solutions to typical few-shot learning problems in a domain while not expressing irrelevant concepts succinctly. This enables, in particular, *synthesis algorithms* to solve learning problems by returning the most succinct expressions in the DSL that satisfy the examples. The design of DSLs which express the right concepts succinctly is a significant engineering challenge, and algorithms for learning DSLs from data can thus provide automation for applying symbolic learning in new domains.

2.1 DSLs for Program Synthesis

The FlashFill program synthesis system [Cambronero et al. 2023; Gulwani 2011] uses a bespoke DSL for expressing common string transformation programs for Excel spreadsheets. The original paper describes the careful design of the DSL and argues that a general-purpose language, like Python, would make the search too complex by “allow[ing] the large number of functions to be combined in unintuitive ways to produce undesirable programs”. Core features of that DSL include an operation $\text{SubStr}(s, P, P)$ for extracting substrings from a string s in a spreadsheet cell, where P is a nonterminal that generates string positions, and an operation $\text{Pos}(R, R, c)$, which generates a string position defined as the c th one whose substring to the left matches a regular expression $r_1 \in L(R)$ and whose substring to the right matches a regular expression $r_2 \in L(R)$.

We seek automatic design of such DSLs given sample learning instances. In this case, operations could be defined using *macros*. For instance, starting from a generic programming language with recursion, the $\text{SubStr}(s, P, P)$ macro could be defined using code which recurses over the input s to find the left position of the substring and return the string ending at the right position.¹

The program synthesis literature is replete with DSLs for various program synthesis tasks—some examples are the small functional programming language on lists defined for the SKETCH-N-SKETCH SVG manipulation framework [Chugh et al. 2016], a DSL designed around an algebra of operators for extracting structured data from text [Le and Gulwani 2014], a DSL for synthesizing barriers for crash consistency of file systems [Bornholt et al. 2016], and a DSL for synthesizing SQL queries from examples [Wang et al. 2017a].

2.2 Synthesis Tools Supporting DSLs

The utility of DSLs for synthesis is reflected in several synthesis tools and frameworks that provide explicit support for defining DSLs and their semantics.

The Rosette and Griset frameworks [Lu and Bodík 2023; Torlak and Bodik 2013] support the definition of solver-aided DSLs, with user-defined syntax and semantics. The syntax-guided synthesis framework [Alur et al. 2015] (SYGUS) supports DSLs with user-specified grammars defining syntax, with semantics often fixed. The SEMGUS framework permits user-specified DSL syntax and semantics [Kim et al. 2021]. Example-based synthesis in SYGUS could be targeted by our formulation of DSL synthesis with grammars, while synthesis for DSLs with semantics defined by new functions could be targeted by our formulation of DSL synthesis with macro grammars.

2.3 Synthesizing Invariants and Feature Engineering

Symbolic learning often requires a set of base features, e.g., in learning decision trees or symbolic regression. For instance, base features might be nonlinear inequalities over numeric variables, with a symbolic learning algorithm considering Boolean combinations over the inequalities. In learning

¹We are motivated by synthesis in domains such as spreadsheet programming, but there are other facets to DSL design we do not consider. For instance, the FlashFill DSL uses “effectively-invertible” top operators and “effectively-enumerable” bottom operators to facilitate *faster* search. Such search performance requirements are out of scope for this paper.

inductive invariants and *specifications* of programs [Astorga et al. 2021; Garg et al. 2014; Zhu et al. 2018], effective techniques have considered Boolean combinations of hand-designed base features.

Consider the GPUVerify tool [Betts et al. 2012] which synthesizes invariants for GPU kernels. It leverages hand-crafted rules for generating basic candidate invariants and then computes the strongest conjunction over these using the Houdini algorithm [Flanagan and Leino 2001]. Useful hand-crafted rules include, e.g., if $i := i \times 2$ occurs in a loop body, then predicates expressing that the loop index i is less than a power of 2 can be useful, e.g., $i < 2$, $i < 4$, $i < 8$, etc., which are common for tree reduction computations. Or, when threads access fixed-size contiguous chunks of a shared array, useful predicates express that the write index is within a bounded region of the array that depends on the thread identifier, e.g., $id \times c \leq write_idx$ and $write_idx < (id + 1) \times c$, where id , c , and $write_idx$ are, respectively, local thread identifiers, constant offsets, and indices where writing occurs in the array. In contrast to designing such predicates by hand, DSL synthesis would seek to automatically discover them from instances of invariant learning problems, which could be sampled from a benchmark of programs to verify. In this setting, we could use a meta-grammar \mathcal{G} to encode the constraint that invariants are *conjunctions* over a set of base predicates, with a DSL synthesis algorithm tasked with determining a good set of base predicates.

DSL synthesis, as proposed in this paper, can serve as a formulation of the feature synthesis problem, with the goal of discovering domain-specific symbolic features using few-shot learning instances drawn from a domain. We can formulate the question as follows: is there a set of n features, each drawn from a class of functions over some existing (even more basic) features, such that a fixed class of symbolic concepts defined over these n features, e.g., specific Boolean combinations, can solve a given set of few-shot learning problems sampled from a domain? Such engineered features can then be used in downstream symbolic learning algorithms for program synthesis [Alur et al. 2015] or symbolic regression [La Cava et al. 2021].

2.4 Library Learning

Library learning is a problem of recent interest [Bowers et al. 2023; Cao et al. 2023; Ellis et al. 2023] related to DSL synthesis. Consider a program synthesizer that solves a class of problems $\mathcal{J} = \{p_1, p_2, \dots\}$ in the program synthesis from examples paradigm. In a library learning phase, we can try to *refactor* existing solutions to some instances p_i in order to learn common concepts— a library L — that enable compression of the set of existing solutions to problems from \mathcal{J} . DREAM-CODER utilizes such refactoring in its *dream phase*, and then, when synthesizing programs for new learning instances, it uses the library L to search for solutions [Ellis et al. 2023]. Recent work has used e-graphs to address library learning with respect to equational theories [Cao et al. 2023].

Library learning is similar to the problem of DSL synthesis with macros introduced in the present work. However, rather than first synthesizing solutions to instances and then asking whether those *particular* solutions can be refactored in a synthesized library, our DSL synthesis problem combines these phases into one— we ask whether there is a library (realized as a macro grammar) such that, for each instance, there is *some* solution expressible using the library. We also introduce the problem of capturing the domain precisely in a DSL, which may be *less expressive* than the base language against which it is defined. Furthermore, we prove decidability results for DSL synthesis problems; previous work on library learning does not provide such results.

2.5 Dimensions of the DSL Synthesis Problem

We collect here the various dimensions of the DSL synthesis problem formulated in this work and summarize their purposes. We discuss these further in Section 4.

1. Expression complexity. DSLs express domain-specific concepts succinctly, i.e., by using symbolic expressions which are syntactically “simple”. We require a formal way to measure the

complexity of a concept as expressed using an expression in a specific DSL. Our results measure simplicity using *parse tree depth* with respect to a DSL, which happens to be relevant in program synthesis, where a common heuristic search technique is to enumerate the language of a grammar by increasing depth. Another natural measure would be *parse tree size*; whether interesting decidability results hold in this case we leave as an open question.

2. Mechanisms for defining DSLs. As discussed earlier, we consider two different mechanisms for defining the syntax of DSLs: regular grammars and macro grammars. Macro grammars are a natural way to express DSLs with functions that take parameters, and furthermore, they are more expressive than regular grammars (we give an example illustrating this in Example 4.3.1).

3. Meta grammar. All variants of the DSL synthesis problems we introduce have a *meta grammar* in the input. Similar to grammar constraints in the context of program synthesis, e.g., in SyGuS [Alur et al. 2015], meta-grammar constraints can express simple kinds of prior knowledge about the space of DSLs one wants to consider. Some simple regular properties that can be encoded with a meta grammar include:

- If our DSL has Boolean formulas, we can ensure any mechanism for defining them can only define monotonic ones, i.e., those for which setting more variables “true” cannot flip the formula from “true” to “false”. It can do so by disallowing atomic formulas that occur under an odd number of negations, for instance.
- Simple kinds of type checking for a finite number of types, e.g., ensuring the generation of Boolean formulas which make comparisons on the values of arithmetic expressions.
- Conjunctive invariants over arbitrary Boolean formulas that define “features” (Section 2.3).

Handling regular syntax constraints makes our results stronger. A user can omit the meta grammar and a synthesizer simply assumes the most permissive meta grammar which adds no constraints.

3 Preliminaries

3.1 Alphabets, Tree Automata, and Tree Macro Grammars

Definition 3.1.1 (Ranked alphabet). A ranked alphabet Δ is a set of symbols with arities given by $\text{arity} : \Delta \rightarrow \mathbb{N}$, $\Delta^i \subseteq \Delta$ is the subset of symbols with arity i , and x^i indicates that symbol x has arity i . We use T_Δ to denote the smallest set of terms containing Δ^0 and closed under forming new terms using symbols of larger arity, e.g. if $t \in T_\Delta$ and $f \in \Delta^1$ then $f(t) \in T_\Delta$. For a set of nullary symbols X disjoint from Δ^0 we write $T_\Delta(X)$ to mean $T_{\Delta \cup X}$. We use *tree* and *term* interchangeably.

We make use of *tree automata* to design algorithms for DSL synthesis. We use *two-way alternating tree automata* which process an input tree by traversing it *up* and *down* while branching universally in addition to existentially. Transitions are given by Boolean formulae which describe valid actions the automaton can take to process its input tree. All automata we deploy have the form $A = (Q, \Delta, Q^i, \delta)$, with states Q , alphabet Δ , initial states $Q^i \subseteq Q$, and δ a transition formula, and they all have acceptance defined in terms of the existence of a run on an input tree. We refer the reader to [Comon et al. 2007, Chapter 7] for background on this presentation of tree automata. We use $\cap_i A_i$ and $\cup_i A_i$ to denote standard constructions of automata that accept intersections and unions of languages for given automata A_i . Finally, note that, given a regular tree grammar G (see [Comon et al. 2007, Chapter 2]), we can compute in polynomial time a non-deterministic top-down tree automaton $A(G)$ with $L(A(G)) = L(G)$.

Definition 3.1.2 (Tree Macro Grammar). A tree macro grammar², or simply *macro grammar*, adds parameters to nonterminal symbols of a regular tree grammar. It is a tuple $G = (S, N, \Delta, P)$, where:

²Tree macro grammars are sometimes called *context-free tree grammars*, e.g., see [Comon et al. 2007, Chapter 2.5]. We prefer *macro grammar* as it evokes macros from programming, a useful intuition when using grammars to define DSLs.

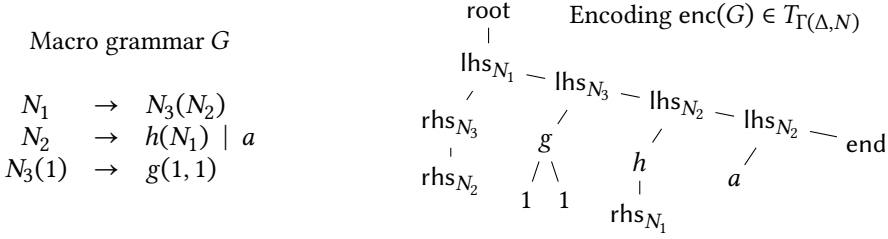


Fig. 1. (Left) Macro grammar G over $\Delta = \{a^0, h^1, g^2\}$ and nonterminals $N = \{N_1^0, N_2^0, N_3^1\}$ and (Right) its encoding as a tree $\text{enc}(G) \in T_{\Gamma(\Delta, N)}$ over grammar alphabet $\Gamma(\Delta, N)$.

N is a finite set of *ranked* nonterminal symbols, $S \in N$ is the starting nonterminal with arity 0, Δ is a finite ranked alphabet disjoint from N , and P is a finite set of rules drawn from $N \times T_{\Delta \cup N}(\mathbb{N})$. We often write rules (N, t) as $N \rightarrow t$ and indicate several rules as usual by $N \rightarrow t_1 \mid \dots \mid t_k$.

We refer to nonterminal symbols with arity greater than 0 as *macro symbols*. When the nonterminal symbols of a macro grammar all have arity 0, i.e. there are no macro symbols, then we recover the standard concept of a *regular tree grammar* as a special case.

We consider only *well-formed* macro grammars in the remainder, i.e. those where right-hand sides of productions refer only to the parameters for the nonterminal on the left-hand side (if any).

Definition 3.1.3 (Well-formed macro grammar). A macro grammar $G = (S, N, \Delta, P)$ is *well formed* if for every $(X, t) \in P$ we have that $t \in T_{\Delta \cup N}(\{1, \dots, \text{arity}(X)\})$.

As usual, we can define the language of a macro grammar to be the set of ground terms derivable in a finite number of steps by applying rules starting from S — but consider the following subtlety.

Example 3.1.1. Consider the macro grammar defined by rules

$$S \rightarrow F(H), \quad F(1) \rightarrow f(1, 1), \quad H \rightarrow a \mid b,$$

where integers indicate the parameters for a macro symbol. Observe that we could apply productions to “outermost” macro symbols first, as in $S \Rightarrow F(H) \Rightarrow f(H, H) \Rightarrow f(a, b)$, or we could apply them “innermost” first, as in $S \Rightarrow F(H) \Rightarrow F(a) \Rightarrow f(a, a)$, or we could mix the two.

Example 3.1.1 illustrates distinct ways to define the language of a macro grammar, and these choices yield different classes of languages [Comon et al. 2007; Fischer 1968]. We consider only outermost derivations and leave an exploration of innermost ones to future work. *Outermost* derivations are those in which rules are never applied to rewrite a nonterminal M if it appears as a sub-term of another nonterminal N . This can be formalized using *contexts* (e.g. see [Comon et al. 2007, Chapters 2.1 and 2.5]) by adding the requirement that any context used in defining the derivation relation must not contain nonterminal symbols.

Definition 3.1.4 (Language of a Tree Macro Grammar). The *language* $L(G) \subseteq T_\Delta$ of a macro grammar G is the set of Δ -terms reachable by applying finitely-many productions in an *outermost order* starting from S . 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, when we say (*macro*) *grammar* we mean *tree (macro) grammar*.

3.2 Encoding Grammars as Trees

We will define tree automata whose inputs are trees that encode grammars. Figure 1 shows an example of how we choose to encode a grammar as a tree; we arrange the grammar rules along the

topmost right-going spine of the tree and use symbols lhs_{N_i} and rhs_{N_i} to distinguish between occurrences of nonterminal N_i on the left-hand and right-hand sides of rules. We use positive integers to indicate the parameters for macro symbols. We write $\Gamma(\Delta, N)$ to denote *grammar alphabets*.

Definition 3.2.1 (Grammar alphabet). Given a ranked alphabet Δ and a set of ranked nonterminal symbols N with maximum macro arity $k \in \mathbb{N}$, we define its *grammar alphabet* as

$$\Gamma(\Delta, N) := \Delta \sqcup \{\text{root}^1, \text{end}^0\} \sqcup \{\text{lhs}_{N_i}^2, \text{rhs}_{N_i}^{\text{arity}(N_i)} : N_i \in N\} \sqcup \{1^0, \dots, k^0\}.$$

Any grammar over Δ using nonterminals N can be encoded as a term over $\Gamma(\Delta, N)$. We define a mapping enc from grammars to the *grammar trees* that encode them, and a mapping dec from grammar trees back to grammars. These are straightforward and can be found in the version with appendix [Krogmeier and Madhusudan 2025]. We elide the distinction between a *grammar* and its encoding as a *grammar tree*.

4 Formulating DSL Synthesis

In this section, we introduce a novel formulation of DSL synthesis that addresses a fundamental aspect of DSLs: namely, a *domain-specific* language should (a) express relevant domain concepts and (b) *not* express irrelevant ones, or at least express them less succinctly than relevant ones. Our formulation is based on *learning*: expressive power of the DSL must be carefully tuned to enable solving an input set of few-shot learning instances. We introduce two distinct mechanisms for specifying which concepts should be expressed in a DSL, one of which requires certain concepts to be expressed, addressing (a), and the other, addressing (b), puts constraints on how succinctly a DSL expresses certain other concepts, if at all.

This section lays the ground for the formal DSL synthesis problems and results developed in Sections 5 to 7. We introduce the two distinct learning signals for our formulation of DSL synthesis and discuss common aspects of all problems studied in this work.

4.1 Learning Instances: Expressive Power and Relative Succinctness

All problems we study involve synthesizing a DSL given *instances of few-shot learning problems*.

Definition 4.1.1 (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.

Learning instances, understood as *inputs* to a DSL synthesizer, contribute *two distinct signals* for construction of a *domain-specific* language, both of which concern expressive power of a DSL. The first signal is about *expressivity* and the second is about *succinctness and inexpressivity* of the DSL. We discuss these in turn using the learning instance depicted in Figure 2, which consists of labeled examples of points in the plane. The training examples $X = P$ are the positive points and the testing examples $Y = N$ are the negative points. In this setting, the *domain* we want to capture (using a DSL) is one where the relevant concepts are sets of points in the plane defined as intersections of a given set of rectangles. In terms of syntax, such a domain might correspond to conjunctions of basic predicates, which is a widespread and useful syntactic bias in various domains, e.g., conjunctive invariants in program verification [Flanagan and Leino 2001].

Expressivity. The first signal says that, given a learning instance we would like to solve, an adequate DSL must have enough power to express *some* concept which solves it³. That is, we want a DSL which contains some expression that satisfies all the examples $X \cup Y$. Consider a DSL for the situation in Figure 2 whose language of expressions consists of Boolean combinations of a fixed

³Similar to related problems like library learning, where either specific given programs must be expressible or a set of learning problems must be solvable using some program (e.g. [Bowers et al. 2023; Cao et al. 2023; Ellis et al. 2023]).

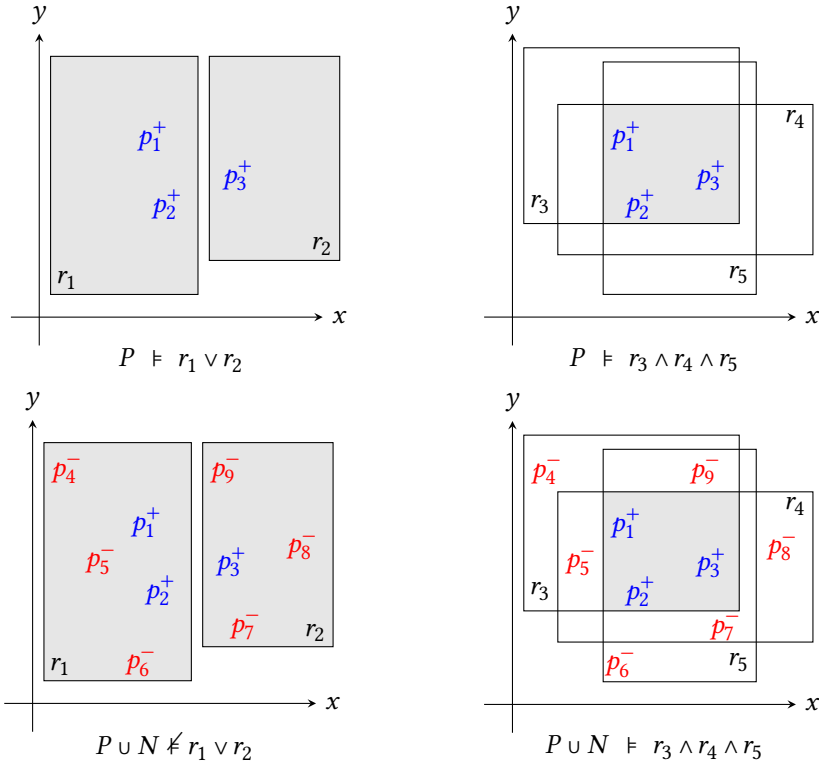


Fig. 2. A learning instance $I = (P, N)$ where examples are points in the plane. Training examples P are the positive points (in blue) and testing examples N are the negative points (in red). Top: the training examples are satisfied by both $r_1 \vee r_2$ and $r_3 \wedge r_4 \wedge r_5$, with r_i signifying membership of a point within the depicted rectangle. Bottom: the testing examples are satisfied by the conjunction but not the disjunction.

set of basic predicates capturing rectangles in the plane, e.g. $r_i := (1 < x < 3) \wedge (2 < y < 4)$. Any DSL containing an expression equivalent to $\varphi := r_3 \wedge r_4 \wedge r_5$ (shown in Figure 2) is adequate because φ satisfies all training examples X (positive points P) and also all the testing examples Y (negative points N). Note that the expressivity signal treats a learning instance as a set of examples $X \cup Y$ and thus forgets about the distinction between training and testing examples.

Relative Succinctness and Inexpressivity. In addition to expressing relevant concepts, a DSL should also *precisely* capture those relevant concepts and perhaps little or nothing else. DSLs need not be fully expressive, and it is in fact a feature if they avoid expressing irrelevant information which does not reflect the domain in question⁴. To address this aspect of DSLs, we formulate a *relative succinctness* constraint which requires that specific concepts should be expressed *less succinctly than other concepts (or not at all)* in a synthesized DSL.

To understand this inexpressivity signal, let us consider Figure 2 once more and address the purpose of splitting the instance into training and testing sets (X, Y) . Recall we have assumed a fixed symbolic language of expressions involving Boolean combinations of some basic rectangles

⁴Imagine a DSL for Excel spreadsheets as in [Gulwani 2011]; there are many irrelevant concepts, like arithmetic on ASCII codes or reversal of strings, which do not reflect typical tasks users of Excel would like to automate.

in the plane. This language is able to represent some regions of the plane using very succinct expressions and for representing other regions it requires less succinct expressions. For this example, let us equate *succinctness* with *syntactic length*, e.g. in Figure 2 the disjunction $r_1 \vee r_2$ is syntactically smaller, and thus more succinct, than the conjunction $r_3 \wedge r_4 \wedge r_5$.

Now, imagine we want to solve the learning instance using a fixed learning algorithm which searches the space of expressible concepts by first considering small expressions and only later considering large expressions if no small expression can be found which is consistent with a small number of examples. Many expression learning algorithms and synthesis tools use such heuristics, with syntax tree size and depth being common measures of succinctness. *For such a fixed algorithm*, along with a given set of learning instances, our goal will be to synthesize a DSL such that learning succeeds *using the fixed algorithm together with the DSL*, and where *success* corresponds to finding an expression that is consistent with *all* the examples in the given learning instance, but where the learning algorithm itself only considers a smaller number of training examples. In other words, we ask for a DSL in which the most *succinct* concepts that it expresses which are consistent with the training examples X are also consistent with the testing examples Y . Learning algorithms which prefer concepts that are simple to express, when operating over such DSLs, will discover solutions which solve training examples *and* which also generalize to testing examples.

Let us return to the situation depicted in Figure 2. Suppose that $\psi := r_1 \vee r_2$ is the shortest expression in our DSL which is consistent with the (positive) training examples X . Since ψ is not consistent with the (negative) testing examples Y , we want to reject this DSL. Afterall, an algorithm with a bias toward succinct expressions will select the smaller ψ instead of the larger, but consistent, $\varphi := r_3 \wedge r_4 \wedge r_5$. The specific learning instance in Figure 2 might favor, for example, a DSL which allows the expression of conjunctions of base rectangles and disallows disjunctions entirely. Our formalization of this inexpressivity signal will in fact also admit DSLs which still express disjunction, but which make it less succinct to express, rather than inexpressible. Thus the formal constraint we introduce (Definition 4.3.9) has to do with the *relative* succinctness of expressions in a language, with one option for satisfying the constraint being to not express certain concepts at all. In the most general case, we are interested in synthesizing DSLs which meet such relative succinctness constraints for *several input learning instances*.

4.2 Base Language and Consistency

In order to synthesize DSLs, we need some mechanism for defining their syntax and semantics. For that purpose, we will assume a specific *base language* with an existing syntax and semantics. We will define the expressions of our DSLs using the syntax of the base language, and the semantics of the DSL expressions is, in this way, inherited from the base language. Furthermore, the base language specifies what counts as an *example* in a learning instance.

Definition 4.2.1 (Base language). The *base language* is specified by a regular tree grammar $G = (S, N, \Delta, P)$, a set \mathcal{M} of *examples*, and a predicate $\text{consistent} \subseteq L(G) \times \mathcal{M}$ that holds when an expression $e \in G$ is consistent with an example $M \in \mathcal{M}$, which we write as $\text{consistent}(e, M)$.

The consistency predicate in Definition 4.2.1 abstracts away details of specific languages while keeping information relevant for DSL synthesis, namely whether a concept expressed in a given symbolic language is consistent with some examples from a domain.

We will identify an abstract *concept* h with a *subset* of examples \mathcal{M}^5 . Given a concept $h \subseteq \mathcal{M}$, we will say an expression e is consistent with h , written $h \models e$, if $\text{consistent}(e, M)$ holds for every $M \in h$ and $\text{consistent}(e, M')$ does not hold for any $M' \in \mathcal{M} \setminus h$.

⁵Concepts can be understood independently of specific symbolic languages we use to express them. For instance, concepts can be arbitrary functions on numbers and DSLs can be specific languages expressing programs that compute the functions.

Example 4.2.1 (Rectangles in the plane). Consider again Figure 2 and suppose we have a fixed and finite set of basic rectangles R . The base language might consist of a grammar like the following

$$S \rightarrow r \in R \mid S \vee S \mid S \wedge S \mid \neg S,$$

with examples $\mathcal{M} = \mathbb{R}^2 \times \{+, -\}$ being labeled points in the plane. Whether an example $((x, y), l)$ is consistent with an expression $\varphi \in L(S)$ is determined by

$$(x, y) \in \llbracket \varphi \rrbracket \quad \text{if} \quad l = + \quad \text{and} \quad (x, y) \notin \llbracket \varphi \rrbracket \quad \text{if} \quad l = -,$$

where $\llbracket \cdot \rrbracket : L(S) \rightarrow \mathcal{P}(\mathbb{R}^2)$ interprets each expression as a subset of the plane in the obvious way. An abstract concept $h \subseteq \mathcal{M}$ in this example corresponds to all labeled points consistent with some subset of the plane, e.g. for $A \subseteq \mathbb{R}^2$ the concept is

$$h_A = \{((a, b), l) : (a, b) \in \mathbb{R}^2, l = + \text{ if } (a, b) \in A \text{ and otherwise } l = -\}.$$

The problems we formulate in Sections 5 to 7 are parameterized by a base language and are thus very general. And our results, as we will see, hold for a large class of base languages.

With the concept of a base language, we can now formalize both the mechanisms for defining DSLs and the constraints captured by learning signals described in Section 4.1.

4.3 DSL Spaces and Properties: Adequacy and Generalization

We formalize DSL synthesis problems along two dimensions: (1) the precise mechanism for specifying DSLs over a base language and (2) properties required of a DSL.

Along dimension (1), we consider specifying DSLs using either grammars or (more expressive) macro grammars defined over the base language. Whether or not macros are allowed, in either case the object we wish to synthesize is a grammar, which, when combined with the base language, satisfies a particular solution concept given by properties (2), which formalize the expressivity and relative succinctness/inexpressivity signals described in Section 4.1 as properties of DSLs which we call *adequacy* and *generalization*. We start with (1) and then address (2).

Definition 4.3.1 (DSL space). Given a base language with grammar $G' = (S', N', \Delta, P')$, the space of DSLs we consider for synthesis is determined by a space of grammars $G = (S, N, \Delta, P)$, with $N' \subseteq N$, which define productions for new nonterminal symbols $N \setminus N'$. Given a synthesized grammar G , the resulting DSL is defined by $\text{extend}(G, G') := (S, N, \Delta, P \cup P')$.

When the DSL space above is determined by tree grammars G we use the phrase *DSL synthesis*. When it ranges over tree *macro* grammars we use the phrase *DSL synthesis with macros*.

Example 4.3.1 (Macro grammars). Macro grammars provide a natural way to model classes of expressions which take parameters, like functions. Regular grammars do not support this, and macro grammars are in fact *more expressive*. The macro grammar below defines a set of expressions—which is *not* regular—encoding functions over two variables x and y . These functions sum the results of applying some other function to each parameter, and the function must be the *same* for each parameter, e.g., things like $h^n(x) + h^n(y)$.

$$S \rightarrow F(x, y), \quad F(1, 2) \rightarrow F(g(1), g(2)) \mid F(h(1), h(2)) \mid 1 + 2$$

Along dimension (2), we consider two properties of DSLs, leading to weak and strong variants of DSL synthesis. In the weak variant, our goal is to synthesize a DSL which, for each input learning instance, expresses some concept that solves the examples it contains. This is the expressivity signal discussed in Section 4.1.

Definition 4.3.2 (Expression solution). Let $I = (X, Y)$ be a learning instance. We say an expression e solves I if it is consistent with $X \cup Y$, and e is consistent with a set of examples X if we have

$$\bigwedge_{M \in X} \text{consistent}(e, M).$$

We write $\text{solves}(e, I)$ and $\text{solves}(e, X)$ when these are true.

We call the weak property required of DSLs *adequacy*.

Definition 4.3.3 (Adequacy). Given a set of learning instances I_1, \dots, I_n , a DSL is *adequate* if, for each I_p , it contains an expression e such that $\text{solves}(e, I_p)$ holds.

The second, and stronger, property we introduce for DSLs corresponds to the relative succinctness of DSL expressions and inexpressivity, as described in Section 4.1. This stronger property is formalized in terms of *concept orderings* induced by DSLs, which depend on specific ways to measure the complexity (or succinctness) of expressions.

Definition 4.3.4 (Expression and concept complexity). *Expression complexity* for a DSL G is defined by a function $c_G : T_\Delta \rightarrow \mathbb{N} \cup \{\infty\}$. Note that expressions are not themselves ordered by such functions, e.g., parse trees for distinct e and e' may have equal depth. Rather, a complexity function partitions expressions and orders cells of the partition. Expression complexity induces a notion of complexity on concepts h by:

$$c_G(h) = \min(\{c_G(e) : e \in T_\Delta, h \models e\}).$$

We single out one particularly common and practically relevant way to measure expression complexity for a DSL: parse tree *depth*⁶. This notion is DSL-dependent, in the sense that an expression e may have a parse of shallow depth in one DSL but only very large depth in another.

Definition 4.3.5 (Parse tree depth complexity). Given a grammar G , we define the depth complexity $\text{depth}_G : T_\Delta \rightarrow \mathbb{N} \cup \{\infty\}$, which for any $e \in G$ is the minimum over the set of all parse trees t for e of the maximum number of nonterminals encountered along any root-to-leaf path in t ⁷. We let $\min(\emptyset) = \infty$. So if there is no parse tree for e in G then $\text{depth}_G(e) = \infty$.

Notions of expression complexity c_G , such as parse tree depth, induce preorders \leq_G on expressions of a DSL G and, more generally, they induce preorders on concepts.

Definition 4.3.6 (Expression and concept orderings). By *expression ordering* or *concept ordering*, we mean in general a preorder on expressions or concepts. We consider expression orderings, written $e \leq_G e'$, that are induced by expression complexity functions c_G which are relative to a DSL G , e.g. $e \leq_G e'$ holds if $c_G(e) \leq c_G(e')$ ⁸. Concept orderings are then induced similarly by induced concept complexity, i.e. $h \leq_G h'$ if $c_G(h) \leq c_G(h')$.

We single out the ordering given by parse tree depth complexity, which appears in our results (Sections 5 to 7).

Definition 4.3.7 (Depth ordering). Given a grammar G and an expression $e \in T_\Delta$, the *depth expression ordering* $e \leq_G e'$ holds when $\text{depth}_G(e) \leq \text{depth}_G(e')$. Similarly, the *depth concept ordering* $h \leq_G h'$ holds when $\text{depth}_G(h) \leq \text{depth}_G(h')$.

The intuition to take away from this is that DSLs organize a space of abstract concepts in a way that captures relevant, domain-specific concepts using symbolic expressions of low complexity.

⁶Many synthesis heuristics explore grammars by increasing depth.

⁷We omit a definition of parse trees; see the version with appendix for an example [Krogermeier and Madhusudan 2025].

⁸The ordering \leq on numbers being the usual one extended to $\mathbb{N} \cup \{\infty\}$.

Example 4.3.2 (Depth concept and expression ordering). Again consider Figure 2 and suppose we have a DSL with an expression grammar G with rules:

$$S \rightarrow r \in R \mid S \wedge S.$$

There are many inexpressible concepts, e.g. those concepts h corresponding to non-convex subsets of the plane such as $\llbracket r_1 \vee r_2 \rrbracket$, for which $\text{depth}_G(h) = \text{depth}_G(r_1 \vee r_2) = \infty$. However, in a more permissive DSL G' , such as one with the following rules

$$S \rightarrow r \in R \mid S \wedge S \mid S \vee S \mid \neg S,$$

which allows all Boolean combinations of rectangles, we have that

$$\text{depth}_{G'}(r_1 \vee r_2) = 2 \leq \text{depth}_{G'}(r_3 \wedge r_4 \wedge r_5) = 3.$$

For the DSL G' , an algorithm which explores expressions in order of increasing depth would find $r_1 \vee r_2$ before it finds $r_3 \wedge r_4 \wedge r_5$, though the latter is consistent with the testing examples in addition to the training examples.

With our notions of expression ordering and complexity measures like depth, we can now state a novel property of DSLs, called *generalization*, which relates to the notions of relative succinctness and inexpressibility described in Section 4.1. The generalization property implies adequacy (i.e. existence of solutions, see Definition 4.3.3), and further requires that some of the most succinct expressions which solve training examples, where succinctness is measured by a fixed expression ordering \leq_G , should also solve testing examples, i.e. some maximally succinct expression that satisfies all training examples should generalize to the testing set. Another way of stating the requirement is that expressions which fail to generalize to testing examples, though they are consistent with training examples, should be relatively no more succinct than an expression which does generalize (or they are not expressed at all).

We call expressions which are consistent with a set of training examples, but *inconsistent* with a set of testing examples, *non-generalizing expressions*.

Definition 4.3.8 (Non-generalizing expression). Given an instance $I = (X, Y)$, a *non-generalizing expression* e is one for which $\text{solves}(e, X)$ holds but $\text{solves}(e, Y)$ does not hold.

Example 4.3.3. The expression $r_1 \vee r_2$ from Figure 2 is non-generalizing because it solves the positive examples (training set X) but does not solve the negative examples (testing set Y).

Finally, we can state the generalization property, which lifts the relative succinctness/inexpressivity requirement to all learning instances given in the input for a DSL synthesis problem.

Definition 4.3.9 (Generalization). Given a set of learning instances I_1, \dots, I_n , a DSL G is *generalizing* for an expression ordering $e \leq_G e'$ if it is adequate (Definition 4.3.3), and additionally, the following holds. For each I_p , there is an expression $e \in G$ such that $\text{solves}(e, I_p)$ holds, and for all $e' \in G$ which are non-generalizing on I_p , we have that $e \leq_G e'$.

We now have the primary concepts needed to introduce our DSL synthesis problems. Before doing so, we discuss (Section 4.4) a natural mechanism for constraining the space of allowed DSLs, namely, regular syntax constraints on grammars, and (Section 4.5) the scope of our forthcoming theorems as it pertains to the base language in terms of which we define DSLs.

4.4 Constraining the DSL Space

The problems we introduce in Sections 5 to 7 are very general. They are parameterized by a base language, which is used to define the syntax and semantics of synthesized DSLs (Section 4.2). In general, the base language should be relatively expressive, as we do not want to assume knowledge

of relevant concepts in the domain, their relationships, and how much power is needed to define them. With an expressive base language, e.g. a Turing-complete programming language, there is no mechanism in the problem specification (though the algorithms we introduce can easily output the *syntactically shortest* DSL solving the problem) which prevents a synthesized DSL from “memorizing” solutions to learning instances by introducing new symbols that exactly define specific solutions, effectively making the solutions into constants and therefore as succinct as possible.

Such memorization can be mitigated by enforcing syntactic constraints on the synthesized DSL. For instance, we might put an upper bound on the number of rules that any specific nonterminal symbol can use, ruling out a grammar like

$$S \rightarrow \text{solution}_1 \mid \dots \mid S \rightarrow \text{solution}_k.$$

Such constraints can be specified in the input, similar to syntax constraints in program synthesis [Alur et al. 2015]. In the DSL synthesis problems we introduce in Sections 5 to 7, syntax constraints are specified in the input using a grammar, which we refer to as a *meta-grammar* because it constrains the syntax of (object) grammars that define DSLs.

Definition 4.4.1 (Meta-grammar). By *meta-grammar* \mathcal{G} we mean a regular tree grammar over a grammar alphabet $\Gamma(\Delta, N)$. We use meta-grammars to constrain the syntax of synthesized DSLs.

Note also that handling a meta-grammar in the input is a *feature* and makes the problems more general, as it can always be omitted and an unconstrained grammar can be used by default⁹.

4.5 Tree Automaton-Computable Language Semantics

Being able to algorithmically check DSLs for adequacy (Definition 4.3.3) and generalization (Definition 4.3.9) implies checking whether a learning instance has *any* solution at all. In other words, for the problem formulations we pursue, being able to verify solutions for DSL synthesis implies that symbolic learning over the underlying base language must be decidable.

Our forthcoming results are *meta-theorems* on the decidability of DSL synthesis. We obtain as specific instantiations of these meta-theorems a swath of decidability results on DSL synthesis over a rich class of base languages recently shown to admit decidable learning [Krogmeier and Madhusudan 2022, 2023], including finite variable logics, modal logics, regular expressions, context-free grammars, linear temporal logic, and some restricted programming languages, among others. Such languages have semantics which can be computed by a tree automaton in the sense that the consistency predicate, when specialized to any *fixed example* M , can be computed by a tree automaton whose size is a function of only the length $|M|$ of an encoding of the example.

Definition 4.5.1 (Tree Automaton-Computable Semantics). Fix a base language consisting of grammar $G = (S, N, \Delta, P)$, examples \mathcal{M} , and predicate $\text{consistent} \subseteq L(G) \times \mathcal{M}$. We say the language semantics *can be evaluated over fixed structures by a tree automaton* if, for any example $M \in \mathcal{M}$, there are computable tree automata A_M and $A_{\neg M}$ that accept, respectively, all expressions consistent with M and all expressions inconsistent with M , i.e., $L(A_M) = \{e \in L(G) : \text{consistent}(e, M)\}$ and $L(A_{\neg M}) = \{e \in L(G) : \neg \text{consistent}(e, M)\}$. We refer to A_M and $A_{\neg M}$ as *example automata*.

In the remainder, we introduce various DSL synthesis problems, varying along the dimensions of (1) plain vs macro grammars and (2) adequacy and generalization as the synthesis specification, and we prove decidability results in each setting.

⁹The version with appendix shows an example of an unconstrained meta grammar [Krogmeier and Madhusudan 2025].

5 Adequate DSL Synthesis

In this section, we introduce the *adequate DSL synthesis problem*, the simplest of the problems we consider. Given a set of few-shot learning instances I_1, \dots, I_l , along with a meta-grammar constraint \mathcal{G} , the requirement is to synthesize an adequate DSL satisfying \mathcal{G} , i.e. one which contains a solution for each instance.

Problem (Adequate DSL Synthesis).

Parameters:

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

Input:

- Instances I_1, \dots, I_l
- Meta-grammar \mathcal{G} over $\Gamma(\Delta, N)$

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

- (1) $\text{extend}(G, G')$ is adequate (Definition 4.3.3) and
- (2) $\text{enc}(G) \in L(\mathcal{G})$, i.e. constraints \mathcal{G} are satisfied

We now establish decidability of adequate DSL synthesis. First we give an overview of the proof and then a more detailed construction after.

5.1 Overview

The proof involves construction of a tree automaton A that reads finite trees encoding grammars. We design A so that it accepts precisely those grammars which are solutions to the problem, with existence and synthesis accomplished using standard algorithms for emptiness of $L(A)$. The main component in the construction is an automaton A_I that accepts grammars that, when combined with the base grammar, are adequate in the sense of Definition 4.3.3: they contain an expression consistent with examples $X \cup Y$, where $I = (X, Y)$. Using these A_I , we then construct a final automaton A that is the product over all A_I and also $A(\mathcal{G})$, an automaton accepting grammars that satisfy the constraint \mathcal{G} . This final automaton A accepts exactly the grammars satisfying \mathcal{G} which contain solutions to each instance I and thus which solve the adequate DSL synthesis problem.

As discussed in Definition 4.5.1, for each example $M \in X \cup Y$, e.g. a labeled point in the plane, we assume existence of a (non-deterministic top-down) *example automaton* A_M with language

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

i.e., the set of expressions in the base language which are consistent with the example. If we can in fact construct such automata for a given base language, then our proof will apply.

The instance automaton A_I reads a grammar tree over alphabet $\Gamma(\Delta, N)$ and explores potential solutions for $I = (X, Y)$ by simulating an automaton A_1 , defined as

$$A_1 := \bigcap_{M \in X \cup Y} A_M, \quad \text{with } L(A_1) = \{e \in L(G') : \text{solves}(e, I)\},$$

which uses the example automata to accept all expressions in the base language that solve I .

Intuitively, A_I operates by walking up and down the input grammar tree to nondeterministically guess a parse tree for an expression e that solves I . 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 navigates to the top spine to find productions corresponding to that nonterminal, and it must

guess which production to use among all those that it finds. 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.

5.2 Automaton Construction

Suppose $A_1 = (Q_1, \Delta, Q_1^i, \delta_1)$. We define a two-way alternating automaton $A_I = (Q, \Gamma(\Delta, N), Q^i, \delta)$. 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 $N_i \neq N_j \in N$, $q \in Q_1$, $x, f \in \Delta$, and $t_1, \dots, t_r \in T_\Delta(\{\text{rhs}_{N_i} : N_i \in N\})$. We use an underscore "_" to describe a default transition when no other case matches.

Mode 1. Find productions. States are drawn from $M1 := (Q_1 \times \{\text{start}\}) \cup (Q_1 \times N)$.

$$\begin{aligned} \delta(\langle q, \text{start} \rangle, \text{root}) &= (\text{down}, \langle q, \text{start} \rangle) & \delta(\langle q, N_i \rangle, \text{lhs}_{N_i}) &= (\text{up}, \langle q, N_i \rangle) \vee (\text{right}, \langle q, N_i \rangle) \\ \delta(\langle q, \text{start} \rangle, \text{lhs}_{N_i}) &= (\text{stay}, \langle q, N_i \rangle) & \delta(\langle q, N_i \rangle, _) &= (\text{up}, \langle q, N_i \rangle) \vee (\vee_{(N_i, \alpha) \in P'} (\text{stay}, \langle q, \alpha \rangle)) \\ \delta(\langle q, N_i \rangle, \text{lhs}_{N_i}) &= (\text{up}, \langle q, N_i \rangle) \vee (\text{left}, q) \vee (\text{right}, \langle q, N_i \rangle) \vee (\vee_{(N_i, \alpha) \in P'} (\text{stay}, \langle q, \alpha \rangle)) \end{aligned}$$

Mode 2. Read productions. States drawn from $M2 := Q_1 \cup (Q_1 \times \text{subterms}(P'))$, where

$$\text{subterms}(P') = \bigcup_{(N_i, \alpha) \in P'} \text{subterms}(\alpha).$$

$$\begin{aligned} \delta(q, x) &= \delta_1(q, x) & \delta(\langle q, \text{rhs}_{N_i} \rangle, _) &= (\text{stay}, \langle q, N_i \rangle) \vee (\vee_{(N_i, \alpha) \in P'} (\text{stay}, \langle q, \alpha \rangle)) \\ \delta(q, \text{rhs}_{N_i}) &= (\text{stay}, \langle q, N_i \rangle) & \delta(\langle q, f(t_1, \dots, t_r) \rangle, _) &= \text{adorn}(t_1, \dots, t_r, \delta_1(q, f)) \end{aligned}$$

The notation $\text{adorn}(t_1, \dots, t_r, \varphi)$ represents a transition formula obtained by replacing each atom of the form (i, q) in the Boolean formula φ by the atom $(\text{stay}, \langle q, t_i \rangle)$ ¹⁰.

Any transition not described by the rules above has transition formula false. The full set of states and the initial states for the automaton are

$$Q := M1 \cup M2, \quad Q^i = \{\langle q, \text{start} \rangle : q \in Q_1^i\} \subseteq M1.$$

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

PROOF. Follows easily by construction [Krogmeier and Madhusudan 2025]. □

5.3 Decidability

We use the construction of A_I to prove the following theorem:

THEOREM 5.2. *The adequate DSL synthesis problem is decidable for any language whose semantics over fixed structures can be evaluated by tree automata (Definition 4.5.1). Furthermore, the set of solutions corresponds to a regular set of trees.*

PROOF. Given meta-grammar \mathcal{G} and instances I_1, \dots, I_l , we construct the product

$$A := A(\mathcal{G}) \cap \text{convert}\left(\bigcap_{p \in [l]} A_{I_p}\right),$$

¹⁰We assume directions in δ_1 are the numbers 1(left), 2(right), 3 ..., etc.

where $\text{convert}(B)$ 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]. By construction and Lemma 5.1 we have

$$L(A) = \{t \in L(\mathcal{G}) : \bigwedge_{p \in [l]} \text{solves}(\text{extend}(\text{dec}(t), G'), I_p)\}.$$

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

COROLLARY 5.3. *Adequate DSL synthesis is decidable in time $\text{poly}(|\mathcal{G}| \cdot \exp(l \cdot m))$, where l is the number of instances and m is the maximum size over all instance automata A_I .*

Remark. The construction of A_I , specifically the simulation of A_1 on a grammar tree, is independent of the learning problem, and it applies essentially unchanged as a proof of the following.

LEMMA 5.4. *Given a tree automaton A , there is a tree automaton B_A that accepts an encoding of a grammar G if and only if $L(A) \cap L(G) \neq \emptyset$.*

PROOF. Follows the same logic as the construction of A_I but leaves out handling a base grammar. \square

6 DSL Synthesis

In this section we introduce the DSL synthesis problem for grammars and prove decidability for ordering based on expression depth. This problem asks for a DSL which orders concepts in such a way that expressions solving learning instances are relatively more succinct than expressions which fail to generalize on testing sets.

Problem (DSL synthesis).

Parameters:

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

Input:

- Instances I_1, \dots, I_l
- Meta-grammar \mathcal{G} over $\Gamma(\Delta, N)$

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

- (1) $\text{extend}(G, G')$ is adequate (Definition 4.3.3) and generalizing (Definition 4.3.9) and
- (2) $\text{enc}(G) \in L(\mathcal{G})$, i.e. constraints \mathcal{G} are satisfied

Solutions to DSL synthesis are grammars that make generalizing expressions appear early in the order and non-generalizing expressions appear later in the order.

We now prove decidability of DSL synthesis over the class of base languages described in Section 4.5 for parse tree depth expression ordering (Definition 4.3.7). The proof has similar structure to that of Section 5, but requires a new idea to construct an automaton that can evaluate arbitrarily large grammars and reason about their induced concept orderings. We introduce the idea with some intuition about *equivalence of grammars*.

6.1 Equivalence of Grammars

If there is to exist an automaton that accepts exactly the grammars solving a DSL synthesis problem, then it must be possible to partition the space of grammars into finitely-many equivalence classes

based on their behavior over an instance I . For *adequate* DSL synthesis the “behavior” of interest was whether or not a grammar expresses at least one solution for each learning problem.

What would make two distinct grammars G_1 and G_2 equivalent with respect to an instance $I = (X, Y)$ under the stronger requirement of generalization (Definition 4.3.9)? Whether G_1 and G_2 are equivalent on I depends on the ease with which they express different concepts relevant to the examples in X and Y . Consider again the example from Figure 2 with $I = (P, N)$. Suppose $G_1 = (S, \{S\}, \Delta, P_1)$ and $G_2 = (S, \{S\}, \Delta, P_2)$, with $\Delta = \{\wedge^2, \vee^2, \neg^1\} \cup \{r^0 : r \in R\}$ and $R = \{r_1, r_2, r_3, r_4, r_5\}$ being a finite set of rectangles in the plane. Suppose the rules P_1 and P_2 are

$$P_1 : S \rightarrow S \wedge S \mid r \in R \quad P_2 : S \rightarrow \neg(\neg S \vee \neg S) \mid r \in R.$$

One way to measure the ease with which G_1 or G_2 expresses concepts is to consider expressible concepts indexed by the depths of the smallest expressions needed to express them. In this case we are interested in which of the examples $P \cup N$ are included in the sets defined by Boolean combinations of rectangles in the plane. To each expression e , we can associate a vector v_e of Boolean values, one per example, which exactly describes how the expression behaves on the instance $I = (P, N)$. For the expression r_1 we have $v_{r_1} = (1, 1, 0, 1, 1, 1, 0, 0, 0)$ because points p_1, p_2 and p_4, p_5, p_6 fall within $\llbracket r_1 \rrbracket$. For any such “behavioral vector”, we want to know the smallest $d \in \mathbb{N}$ for which it is expressible using an expression of depth d but no shallower. This depth depends on the grammar. We can encode such information in grammar-specific tables whose entries are subsets of behavioral vectors and whose columns and rows are indexed by nonterminals and increasing integers, respectively, as depicted in Figure 3.

| | | | | | | | | |
|---|---------------------------------|----------|---|---|--------|---|--|--|
| $G_1 : S \rightarrow S \wedge S \mid r \in R$ | | | $G_2 : S \rightarrow \neg(\neg S \vee \neg S) \mid r \in R$ | | | $G_3 : S \rightarrow S \wedge S \mid S \vee S \mid r \in R$ | | |
| d | S | | d | S | | d | S | |
| 1 | $\{v_{r_1}, \dots\}$ | | 1 | $\{v_{r_1}, \dots\}$ | | 1 | $\{v_{r_1}, \dots\}$ | |
| 2 | $\{v_{r_1 \wedge r_2}, \dots\}$ | \equiv | 2 | $\{v_{\neg(\neg r_1 \vee \neg r_2)}, \dots\}$ | \neq | 2 | $\{v_{r_1 \vee r_2}, \dots\}$ | |
| \vdots | \vdots | | \vdots | \vdots | | 3 | $\{v_{r_3 \wedge (r_4 \wedge r_5)}, \dots\}$ | |
| | | | | | | \vdots | \vdots | |

Fig. 3. Tables for different grammars over the instance $I = (P, N)$ from Figure 2 which capture expressive power relative to I , stratified by parse tree depth d . Tables for G_1 and G_2 are identical. The table for G_3 is distinct, and registers a non-generalizing expression $r_1 \vee r_2$ before the first generalizing one $r_3 \wedge r_4 \wedge r_5$.

To understand the tables in Figure 3, consider the simultaneous least fixpoint that defines, for instance, $L(G_1)$ as a set of Δ -terms. Though it is an infinite set, if we consider terms modulo equivalence relative to examples in (P, N) , then there are finitely-many equivalence classes— with the 9 points from Figure 2 there are 2^9 classes— and the fixpoint computation needs no more than that number of steps to terminate. Beyond some depth $k \leq 2^9$, expressions of G_1 or G_2 repeat themselves with regard to $I = (P, N)$. The tables in Figure 3 display classes at their earliest achievable depth— the entry at row i and column j contains the set of behavioral vectors achieved first at depth i for nonterminal j (in this case there is only a single nonterminal S).

It is easy to see that the tables for G_1 and G_2 are in fact identical, since $\varphi \wedge \varphi'$ is logically equivalent to $\neg(\neg\varphi \vee \neg\varphi')$ for any formulas φ and φ' . In general, whether or not two syntactically distinct grammars correspond to identical tables depends on the instance $I = (X, Y)$, though in the specific case of G_1 and G_2 they have identical tables for any I whatsoever.

If the “behavioral vectors” are drawn from a finite set for any fixed instance I , then we can argue that the rows of these tables must repeat after a certain finite depth for any learning instance.

We can then ask which among the finitely-many bounded-depth tables over this domain a given grammar corresponds to, and this gives us a finite index equivalence relation for grammars. We will design automata which read grammars (presented as trees) and check which class a grammar corresponds to by iteratively computing its table row by row. Whether a grammar satisfies the generalization constraint (Definition 4.3.9) for a specific instance I can be determined by checking whether a non-generalizing expression is encountered at an earlier row in the table than any generalizing one. Consider the table for G_3 in Figure 3, which is distinct from the tables for G_1 and G_2 . It includes $v_{r_1 \vee r_2} = (1, 1, 1, 1, 1, 1, 1, 1, 1)$ at depth 2, which does not appear in any row of the other tables, and it includes $v_{r_3 \wedge (r_4 \wedge r_5)}$, a vector for a generalizing expression, only at depth 3. We would want to reject G_3 for this reason.

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. We use the information captured by such tables, albeit with a more complex domain D , in our automaton construction for the proof of decidability.

Simulating a Dynamic Program Using a Tree Automaton. Computing such a table for a given grammar G can be accomplished with a *dynamic program* that computes the rows starting from index 0 up to a bound which depends on each instance (X, Y) . We in fact use a slight modification of the tables in Figure 3 which take a union of the entries from previous rows in later rows. We refer to these as *recursion tables*. Given G , a dynamic program can compute the entry for a non-terminal N in the recursion table for a given row by taking the union of all values achievable by G using productions for N , with nonterminals in the right-hand sides of the productions interpreted using values achieved in earlier rows.

For instance, if the program is running for grammar G_1 from Figure 3 and computing the entry of the table T at row 2 and column S , it computes

$$T[2, S] := T[1, S] \cup \{v \wedge v' : v, v' \in T[1, S]\} \cup \{(\mathbb{1}_{[r]}(p_1), \dots, \mathbb{1}_{[r]}(p_9)) : r \in R\},$$

where \wedge above indicates a component-wise conjunction on vectors.

Notice, however, that for the DSL synthesis problem we do not have a grammar G over which to run this dynamic program— the grammar is what we want to synthesize. What we will in fact do is simulate the dynamic program over a grammar encoded as input to a tree automaton. In order to simulate this algorithm accurately, the automaton will nondeterministically guess the values for each row and verify the guesses by walking up and down the input grammar and simulating the example automata which can be used to determine existence of expressions in the grammar which correspond to specific behavioral vectors for the learning examples.

We now describe the details of this construction further.

6.2 Automaton Construction

The main component of our construction is an automaton A_I accepting grammars that, when combined with the base grammar, solve an instance $I = (X, Y)$. Our final automaton A will involve a product over the instance automata A_I .

Similar to the construction in Section 5, we assume existence of *example automata* (Definition 4.5.1). For each example $M \in X \cup Y$, we assume non-deterministic top-down tree automata A_M and $A_{\neg M}$ over alphabet Δ with languages

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

We can now define *instance automata* A_1^I and A_2^I :

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

We omit the superscript and write A_1 or A_2 when the instance I is clear. The automaton A_1 accepts all generalizing expressions and the automaton A_2 accepts all non-generalizing expressions:

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

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

Suppose $A_1 = (Q_1, \Delta, Q_1^i, \delta_1)$ and $A_2 = (Q_2, \Delta, Q_2^i, \delta_2)$. Note that, as constructed, A_1 and A_2 are non-deterministic top-down automata. We will consider tables similar to those described in Section 6.1 whose entries range over the powerset $\mathcal{P}(Q_1 \sqcup Q_2)$. On an input grammar tree, our automaton A_I will iteratively construct the rows of its corresponding *recursion table* for I .

Recursion Tables. 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, \dots, N_k , with $N_1 = S$. Now let $H_i : \mathcal{P}(Q_1 \sqcup Q_2)^k \rightarrow \mathcal{P}(Q_1 \sqcup Q_2)^k$ be the operator defined by the equation

$$H_i(R) = \bigcup_{(N_i, t) \in P} \llbracket t \rrbracket_R^{A_1} \sqcup \llbracket t \rrbracket_R^{A_2}, \quad R \in \mathcal{P}(Q_1 \sqcup Q_2)^k.$$

The notation $\llbracket t \rrbracket_R^{A_j}$, for $j \in \{1, 2\}$, denotes the subset of Q_j reachable¹¹ by running the automaton

$$A'_j = (Q_j, \Delta \sqcup \{\text{rhs}_{N_s} : N_s \in N\}, Q_j^i, \delta'_j)$$

on term t , where $\delta'_j(q, \text{rhs}_{N_s}) = \text{true}$ for each $q \in R_s \cap Q_j$ 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 $H : \mathcal{P}(Q_1 \sqcup Q_2)^k \rightarrow \mathcal{P}(Q_1 \sqcup Q_2)^k$ defined by $H(R) = (H_1(R), \dots, H_k(R))$ 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:

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

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 $T(G)[i, j]$ ¹², consists of the subset of values from $Q_1 \sqcup Q_2$ that are first achieved at parse tree depth i for nonterminal N_j . For $1 \leq j \leq k$:

$$T(G)[0, j] := \emptyset \quad \text{and} \quad T(G)[i, j] := H^i(Z_0)_j \setminus H^{i-1}(Z_0)_j, \quad \text{for } 0 < i \leq n^*.$$

By construction, for the depth concept ordering (Definition 4.3.7) given by $\text{depth}_G(e) \leq \text{depth}_G(e')$, a grammar G solves $I = (X, Y)$ if and only if Equation (1) holds:

$$\text{Exists a row } i \text{ such that } F_1 \cap T(G)[i, 1] \neq \emptyset \text{ and for all } 0 \leq j < i, F_2 \cap T(G)[j, 1] = \emptyset \quad (1)$$

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 $T(G)$ is *acceptable* if this holds.

¹¹The subset of states starting from which the automaton has an accepting run on t .

¹²For convenience, we index rows starting from zero and columns starting from one.

We define a tree automaton A_I whose language is

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

On input $t \in T_{\Gamma(\Delta, N)}$, the automaton iteratively guesses the row-by-row construction of the recursion table $T_t := T(\text{extend}(\text{dec}(t), G'))$ starting from row 0 and working downward to row n^* . At each increasing depth d , it keeps track of which domain values have not yet been achieved and guesses which new ones can be achieved in depth d using previously computed values at depths less than d . It verifies the guesses by simulating instance automata A_i on the right-hand sides of grammar rules for each nonterminal. As it constructs the recursion table, it simultaneously checks that the table is acceptable according to Equation (1) and accepts or rejects accordingly.

The details of the construction can be found in the version with appendix [Krogmeier and Madhusudan 2025]. We note that the number of states for A_I is exponential in the sizes of the instance automata, as the entries of the recursion table range over subsets of their states.

6.3 Decidability of DSL Synthesis

THEOREM 6.1. *DSL synthesis is decidable with depth concept ordering (Definition 4.3.7) for any language whose semantics on any fixed structure can be evaluated by tree automata (Definition 4.5.1). Furthermore, the set of solutions corresponds to a regular set of trees.*

PROOF. After construction of A_I the proof is identical to Section 5.3. \square

COROLLARY 6.2. *For languages covered by Theorem 6.1, DSL synthesis with depth ordering is decidable in time $\text{poly}(|\mathcal{G}| \cdot \exp(l \cdot \exp(m)))$, where l is the number of learning instances and m is the maximum size over all instance automata.*

To make the content of Theorem 6.1 more explicit, consider an instance of DSL synthesis which is decidable as a result. Finite-variable first order logic can be evaluated by tree automata in the sense of Definition 4.5.1. In particular, this means that, given a base language consisting of first-order logic over, e.g., finite graphs, with formulas working with finitely-many variables, the DSL synthesis problem with depth ordering is decidable. The tree automata for evaluating logic formulas have size exponential in the number of examples $s = |X| + |Y|$ for a learning instance (X, Y) consisting of Boolean-labeled graphs, exponential in the size n of graphs $G \in X \cup Y$, and doubly exponential in the number of variables k that are allowed in formulas. So by Corollary 6.2 we have that DSL synthesis with depth ordering for finite-variable first order logic over finite graphs is decidable in time $\text{poly}(|\mathcal{G}| \cdot \exp(l \cdot \exp(m)))$, where $m(n, s, k) = \exp(ns^k)$. Similar results follow immediately for several other languages, e.g., those from [Krogmeier and Madhusudan 2023].

Remark. The construction of A_I in Section 6.2, specifically the simulation of A_1^I and A_2^I on the grammar input, is independent of the learning problem and can be used to prove the following.

LEMMA 6.3. *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 $L(G)$.*

PROOF. Follows the same logic as the construction of A_I with some simplifications. \square

Open Problem. Finally, we leave open the question of whether an analogous result to that of Theorem 6.1 holds when the concept ordering is given by parse tree size rather than depth. It is unlikely that the solution sets would be regular sets, as this would seem to imply the regularity of sets such as $\{f(t, t) : t \text{ an arbitrarily large term}\}$, which are not in fact regular, though the existence of suitable DSLs may still be decidable.

7 DSL Synthesis for Macro Grammars

We now introduce variants of the problems from Sections 5 and 6 which define DSLs using *macro grammars* (see Section 3.1 and Example 4.3.1). We establish decidability results for each variant.

7.1 Adequate DSL Synthesis with Macros

Problem (Adequate DSL Synthesis with Macros).

Parameters:

- Finite set of nonterminals N containing some macro symbols
- Base language $G' = (S', N', \Delta, P')$ with $N' \subseteq N$ // a regular tree grammar

Input: Instances I_1, \dots, I_l and meta-grammar \mathcal{G} over $\Gamma(\Delta, N)$

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

- (1) $\text{extend}(G, G')$ is adequate (Definition 4.3.3) and
- (2) $\text{enc}(G) \in L(\mathcal{G})$, i.e. constraints \mathcal{G} are satisfied

THEOREM 7.1. *Adequate DSL synthesis with macros is decidable for any language whose semantics over fixed structures can be evaluated by tree automata (Definition 4.5.1). Furthermore, the set of solutions corresponds to a regular set of trees.*

Macros lead us to a more complex decision procedure for DSL synthesis— we briefly explain the adjustments needed to prove decidability— a complete construction can be found in the version with appendix [Krogmeier and Madhusudan 2025].

The proof of Theorem 7.1 is similar to that of adequate DSL synthesis from Section 5, except A_I uses exponentially more states to deal with macros. To simulate the instance automaton A_I , it keeps track of *sets* of distinct expressions generated by a given nonterminal. To see why, consider the grammar with rules $S \rightarrow H(G)$, $H(1) \rightarrow h(1, 1)$, and $G \rightarrow a \mid b$. Imagine A_I is reading $S \rightarrow H(G)$ and checking that S generates an expression evaluating to $q \in Q_1$, a state of A_I . It might check that an expression generated in H can evaluate to q , *assuming* that the argument G generates an expression evaluating to some $q_G \in Q_1$. Notice, however, that $H(G) \Rightarrow h(G, G) \Rightarrow h(a, b)$ is a valid outermost derivation, and G generated two distinct expressions despite being passed once as an argument to H . The automaton can handle this by tracking *the entire subset* of Q_1 that expressions generated by G can evaluate to. Besides this increase in states to handle macros, the construction and decision procedure are similar to that of Section 5.

7.2 DSL Synthesis with Macros

Here we define DSL synthesis with macros and state a decidability result for depth ordering.

Problem (DSL Synthesis with Macros).

Parameters:

- Finite set of nonterminals N containing some macro symbols
- Base language $G' = (S', N', \Delta, P')$ with $N' \subseteq N$ // a regular tree grammar
- Expression ordering \leq

Input: Instances I_1, \dots, I_l and meta-grammar \mathcal{G} over $\Gamma(\Delta, N)$

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

- (1) $\text{extend}(G, G')$ is adequate (Definition 4.3.3) and generalizing (Definition 4.3.9) and
- (2) $\text{enc}(G) \in L(\mathcal{G})$, i.e. constraints \mathcal{G} are satisfied

The new challenge in this setting is to account for an interaction between the relative succinctness constraint (Definition 4.3.9) and potentially deep nesting of macro applications. Our result below holds for classes of macro grammars where all grammar rules have macro application nesting depths bounded by a constant. Details can be found in the version with appendix [Krogmeier and Madhusudan 2025].

THEOREM 7.2. *DSL synthesis with macros is decidable for depth ordering over any language whose semantics on fixed structures can be evaluated by tree automata (Definition 4.5.1) for any class of macro grammars whose macro nesting depth is bounded. Furthermore, the set of solutions corresponds to a regular set of trees.*

Suppose we have a meta grammar \mathcal{G} . If there is a bound $b \in \mathbb{N}$ on the nesting depth of macros occurring in any grammar rule of any grammar in \mathcal{G} , then we can compute b given \mathcal{G} and use it in an automaton construction similar to that of Section 6.3 to synthesize and decide existence of DSLs with macros which abide by the meta grammar constraint.

Given $b \in \mathbb{N}$, we adapt the construction of A_I from Section 6.3. In order to compute the values achievable by nonterminals at a given depth in the presence of nested macros, A_I now keeps track of previously computed rows of the table individually, rather than keeping track of a *union* of previously computed rows. Because the nesting depth of macros is bounded by b , the automaton need only remember b previous rows in order to accurately compute the table entries. Additionally, for macro symbols, the entries of the table correspond to *functions* on sets of values rather than sets. Besides these differences the construction is similar to Section 6.3.

Open Problem. Does a result analogous to Theorem 7.2 hold for the case of unbounded macro nesting depth? If the solution sets are not regular, is the problem at least decidable?

8 Related Work

Program synthesis and library learning. Hand-designed DSLs are crucial in many applications of example-based program synthesis, e.g. [Bornholt et al. 2016; Chugh et al. 2016; Le and Gulwani 2014; Wang et al. 2017a]. Excel’s FlashFill [Gulwani 2011] enabled program synthesis from few examples for spreadsheet programming, and a major factor in its success was a DSL that succinctly captured useful spreadsheet operations. Not only do DSLs define specific domains, but they can also make synthesis more tractable by reducing search space sizes. Tradeoffs between expressive grammars and synthesizer performance were studied for SyGuS problems in [Padhi et al. 2019].

Work on library learning explores the problem of compressing a given corpus of programs [Bowers et al. 2023; Cao et al. 2023] or refactoring knowledge bases expressed as logic programs [Dumancic et al. 2021], where the goal is to find a set of programs which can be composed to generate the input corpus or which are logically similar or equivalent to the input, but which also serves to compress it. This contrasts with our work because DSL synthesis does not assume a given set of solutions and instead requires a DSL that expresses succinct solutions to given learning problems. Our formulation also does not require an algorithm to solve the hard problem of finding semantically *equivalent* representations of a particular set of solutions. Closer in spirit to our work are the EC² and DREAMCODER systems [Ellis et al. 2018, 2023], which learn a library alongside program search to solve classes of synthesis problems from specific domains. In contrast to our work, these systems cannot declare there is *no library* over a given signature which solves a set of learning problems. We also introduce a new signal related to the *relative succinctness* of DSL concepts, and our formulation permits expressive power of the base language to *decrease*.

Learning grammars and automata. A large body of work explores the learning of formal languages, e.g., L^* [Angluin 1987] and RPNI [Oncina and García 1992] learn regular languages represented by automata. Recent work studies context-free grammar learning for data format discovery

and fuzz testing [Bastani et al. 2017; Jia and Tan 2024; Kulkarni et al. 2021; Miltner et al. 2023]. In these applications, useful grammars express syntactic properties, e.g., well-nested XML. This is very different from DSL synthesis, where the relevant grammar properties depend on language *semantics* and are determined by given learning instances. Similarly distinct recent work synthesizes grammars capturing programs that implement specific exploits [Ling et al. 2025].

Vanlehn and Ball [Vanlehn and Ball 1987] approached grammar learning using version space algebra [Mitchell 1982]. The automata in our proofs represent version spaces: each automaton represents the set of all grammars which express solutions for some specific learning instance.

Applications of tree automata to synthesis. Tree automata underlie several classic 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]; such automata read trees encoding system behaviors, whereas the automata in our work read encodings of DSLs and the parse trees they admit. Our automata for DSLs rely on existing constructions which, given an arbitrary finitely-presented structure, produce a tree automaton that reads parse trees of expressions in the base language and evaluates them over that structure using memory that is independent of the expression size but may depend on the structure. These constructions were used for recent decidability results for learning in finite-variable logics [Krogmeier and Madhusudan 2022] and other symbolic languages [Krogmeier and Madhusudan 2023]; our results apply to DSL synthesis over all such languages. Related techniques have been used for decidability results in program synthesis, e.g., reactive programs from temporal logic specifications [Madhusudan 2011] and uninterpreted programs from sketches [Krogmeier et al. 2020], and as an algorithmic framework in program synthesis tools, e.g., [Gulwani 2011; Polozov and Gulwani 2015; Wang et al. 2017b, 2018].

9 Conclusion

We introduced the problem of synthesizing DSLs from few-shot learning instances. Our formulation contributes a new *relative succinctness* constraint on synthesized DSLs, which requires them to capture a domain precisely by (a) expressing domain-specific concepts using succinct expressions and (b) expressing irrelevant concepts using only less succinct ones, or perhaps not expressing them at all. DSLs are represented using (macro) grammars which are defined over a base language that gives semantics to the symbols in the DSL. The DSL synthesis problems we introduce, and the relative succinctness constraint especially, emphasize the automated construction of DSLs *for few-shot synthesis*; they ask for DSLs using which specific synthesis algorithms succeed.

We proved that DSL synthesis is decidable when succinctness corresponds to small parse tree depth, and the solutions sets (i.e., DSLs) correspond to regular sets of trees. The result holds for a rich class of base languages whose semantics over any fixed structure can be evaluated by a finite tree automaton. We also proved decidability for variants of the DSL synthesis problem where (a) the relative succinctness constraint is replaced by a weaker one requiring only that DSLs express some solution for each instance and (b) where DSLs are defined using grammars with macros.

Future work should explore practical implementations of DSL synthesis and probe whether DSLs can indeed be realized by synthesis from few-shot learning problems and, if so, how much data is needed to arrive at useful DSLs in specific domains.

Data Availability Statement

This paper has no accompanying artifact.

Acknowledgments

This work was supported in part by a Discovery Partners Institute (DPI) science team seed grant and a research grant from Amazon.

References

- Rajeev Alur, Rastislav Bodik, Eric Dallar, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2015. Syntax-Guided Synthesis. In *Dependable Software Systems Engineering*, Maximilian Irlbeck, Doron A. Peled, and Alexander Pretschner (Eds.). NATO Science for Peace and Security Series, D: Information and Communication Security, Vol. 40. IOS Press, 1–25. <https://doi.org/10.3233/978-1-61499-495-4-1>
- 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](https://doi.org/10.1016/0890-5401(87)90052-6)
- Angello Astorga, Shambwaditya Saha, Ahmad Dinkins, Felicia Wang, P. Madhusudan, and Tao Xie. 2021. Synthesizing contracts correct modulo a test generator. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 104 (oct 2021), 27 pages. <https://doi.org/10.1145/3485481>
- 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>
- Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: a verifier for GPU kernels. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 113–132. <https://doi.org/10.1145/2384616.2384625>
- James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. 2016. Specifying and Checking File System Crash-Consistency Models. (2016), 83–98. <https://doi.org/10.1145/2872362.2872406>
- 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 Cachet. 2002. *Two-Way Tree Automata Solving Pushdown Games*. Springer-Verlag, Berlin, Heidelberg, 303–317.
- José Cambronero, Sumit Gulwani, Vu Le, Daniel Perelman, Arjun Radhakrishna, Clint Simon, and Ashish Tiwari. 2023. FlashFill++: Scaling Programming by Example by Cutting to the Chase. *Proc. ACM Program. Lang.* 7, POPL, Article 33 (Jan. 2023), 30 pages. <https://doi.org/10.1145/3571226>
- 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>
- Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and direct manipulation, together at last. *SIGPLAN Not.* 51, 6 (June 2016), 341–354. <https://doi.org/10.1145/2980983.2908103>
- 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.
- Sebastijan Dumancic, Tias Guns, and Andrew Cropper. 2021. Knowledge Refactoring for Inductive Program Synthesis. *Proceedings of the AAAI Conference on Artificial Intelligence* 35, 8 (May 2021), 7271–7278. <https://doi.org/10.1609/aaai.v35i8.16893>
- Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. 2018. Learning Libraries of Subroutines for Neurally-Guided Bayesian Program Induction. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2018/file/7aa685b3b1dc1d6780bf36f7340078c9-Paper.pdf
- 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>
- Michael J. Fischer. 1968. Grammars with macro-like productions. In *9th Annual Symposium on Switching and Automata Theory (swat 1968)*. 131–142. <https://doi.org/10.1109/SWAT.1968.12>
- Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity (FME '01)*. Springer-Verlag, Berlin, Heidelberg, 500–517.
- Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014. ICE: A Robust Framework for Learning Invariants. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 69–87.

- 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>
- Xiaodong Jia and Gang Tan. 2024. V-Star: Learning Visibly Pushdown Grammars from Program Inputs. *Proc. ACM Program. Lang.* 8, PLDI, Article 228 (June 2024), 24 pages. <https://doi.org/10.1145/3656458>
- 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 and P. Madhusudan. 2025. Synthesizing DSLs for Few-Shot Learning. arXiv:cs.PL/2508.16063 <https://arxiv.org/abs/2508.16063>
- 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>
- William La Cava, Patryk Orzechowski, Bogdan Burlacu, Fabricio de Franca, Marco Virgolin, Ying Jin, Michael Kommenda, and Jason Moore. 2021. Contemporary Symbolic Regression Methods and their Relative Performance. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, J. Vanschoren and S. Yeung (Eds.), Vol. 1. Curran. <https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c0c7c76d30bd3dcaefc96f40275bdc0a-Abstract-round1.html>
- 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>
- Yuxi Ling, Gokul Rajiv, Kiran Gopinathan, and Ilya Sergey. 2025. Sound and Efficient Generation of Data-Oriented Exploits via Programming Language Synthesis. In *34th USENIX Security Symposium (USENIX Security 25)*. USENIX Association, Seattle, WA, 413–429. <https://www.usenix.org/conference/usenixsecurity25/presentation/ling>
- Sirui Lu and Rastislav Bodík. 2023. Griset: Symbolic Compilation as a Functional Programming Library. *Proc. ACM Program. Lang.* 7, POPL, Article 16 (Jan. 2023), 33 pages. <https://doi.org/10.1145/3571209>
- 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. Sagittarius: A DSL for Specifying Grammatical Domains. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 293 (Oct. 2023), 29 pages. <https://doi.org/10.1145/3622869>
- Tom M. Mitchell. 1982. Generalization as search. *Artificial Intelligence* 18, 2 (1982), 203–226. [https://doi.org/10.1016/0004-3702\(82\)90040-6](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.
- 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>
- Emina Torlak and Rastislav Bodik. 2013. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 135–152. <https://doi.org/10.1145/2509578.2509586>
- 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, 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. 2018. Relational Program Synthesis. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 155 (Oct. 2018), 27 pages. <https://doi.org/10.1145/3276525>
- He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A data-driven CHC solver. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 707–721. <https://doi.org/10.1145/3192366.3192416>

Received 2024-10-16; accepted 2025-08-12